# Ada* Verification System (AVS) Studies

Final Report

Prepared for:
Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, VA 22090-5500

Prepared by:
IIT Research Institute
4550 Forbes Blvd., Suite 300
Lanham, MD 20706

November 1987

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

## PREFACE

IITRI is quite pleased with the Final Report. In addition to this report we are passing along, through this forward, several comments made by one of the reviewers of the draft of this document.

Our purpose in passing these concerns along to you is to specifically indicate related issues that are either not addressed or are not completely investigated by this study. In addition, there are instances noted where the perspective taken by IITRI is not universally shared throughout the software community.

One area of concern raised was the relative treatment of formal code verification, software safety, and IBM's 'cleanroom.' The Orange Book discussion of Beyond Class (A1) accepts code verification as an approach for these systems but does not rule out other approaches. The report does not fully develop the argument that formal code verification raises the assurance of software to a higher level than the other two approaches.

Further, a feasibility assessment of applying the alternative approaches to a large system versus applying formal code verification to a large system was not performed. The point raised is that 'cleanroom' has been successfully applied to these systems while formal code verification has not. This could lead to the conclusion that although 'cleanroom' may not raise assurance to the extent that formal code verification does, the feasibility of applying 'cleanroom' to large systems has been established.

The assessment of verifiability of individual Ada constructs is a necessary step; the issue of interaction effects between constructs relative to verification must be addressed prior to designing or implementing a verification environment. Also, further detail does exist on specific constructs. In particular, the field of numerical analysis has developed approaches toward improving the verifiability of programs using real numbers. Although the use of real numbers goes beyond the Ada language, it is certainly applicable to Ada.

Other issues which require ongoing research include the scope and assumptions of the verification. Do you stop with the source code or also verify all runtime support software? Do you verify the compiler? Do you interpret the semantics for the source code from the support code necessary for execution? Do you assume normal termination of the program? Which constructs are more sensitive to abnormal termination?

This study investigated issues relative to the use of Ada for systems to be certified Beyond Class (A1). There are specific sections of the Orange Book which address systems at the Class (A1) level and below which are affected by the implementation language. This report does not address those issues.

Our resource estimate could have used additional systems as analogs, and gone into more detail on the similarities and differences of the existing systems to the desired Ada verification environment.

Finally, all of us would like to have more detail on the ongoing efforts.

Again, we feel very pleased with the study and the Final Report. We also feel it is in everyone's best interest to pass along these concerns.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1. REPORT NUMBER / 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) Ada* Verification System (AVS) Studies--Final Report | 5. TYPE OF REPORT & PERIOD COVERED |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) IIT Research Institute | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION AND ADDRESS IIT Research Institute 4600 Forbes Blvd. Lanham, MD 20706 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081 | 12. REPORT DATE November, 1987 |
| | 13. NUMBER OF PAGES 65 |
| 14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) | 15. SECURITY CLASS (of this report) UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Verification System, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of this study was to investigate approaches to 'high-assurance' software written in the Ada programming language. 'High-assurance' software is an expression used throughout this report that includes the software in systems defined to be secure by the Department of Defense Trusted Computer System Evaluation Criteria as well as other software with very high reliability or security requirements. 'High-assurance' software includes any software which must function as intended or there would be a threat to human life or national security. The report is applicable, then, to high-assurance software which, for the most part, is yet to be developed.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. The techniques and technologies investigated by this study are applicable to systems to be certified as 'Beyond Class (A1), as well as to other high-assurance software. The primary approach in developing this software is formal code verification. This report investigates each Ada construct relative to code verificatio, the technologies necessary to support code verification, ongoing efforts direetly related to verification of Ada code, and alternatives to code verification for achieving high-assurance software.

Since Ada was not developed to be a verifiable language, there are some constructs that will defy formal verification; these challenges do not seem to be overwhelming and could presumably be controlled by restrictions on the use of the language. Tasking and exception handling are the two greatest challenges that the language constructs provide for verification, with tasking being the greater challenge.

The two most critical support technologies for Ada code verification are a formal definition of the language and a formal specification language. The formal definition of Ada developed by DDC needs to be verified, validated, or certified by someone outside of the developing group. This is a major issue. Also, the structure and the syntax of the definition limit its utility. ANNA as a specification language has limitations which are being addressed by Odyssey Research Associates, and alternative forms for specifications are being investigated by Computatuional Logic.

Two alternatives to formal verification were investigated, the software safety approach and the IBM 'cleanroom'. Each improves the assurance of software yet neither provides adequate assurance to be considered for 'Beyond Clas (A1)' software.

Assessing the state of the art of formal verification technology relative to the Ada language requires perspective. To date, the largest code verified system in operation is 4,211 lines of code. Given that languages that are designed and developed to be verifiable provide challnges to the development of large, complex systems, it would be naive to expect that Ada would be easibly verifiable.

There are a few general conclusions that have been develped during the course of this study:

Reasons not to use Ada at the Class (A1) level or below are more culturally based than technically based.

Formality in software development should not be all or nothing.

Analysis of constructs that challenge verification can be the basis for developing coding guidelines on software that do not need to be verified.

There are very few people who adequately understand the application of formal methods.

TABLE OF CONTENTS

## 0.0 EXECUTIVE SUMMARY

The goal of this study was to investigate approaches to 'high-assurance' software written in the Ada programming language. 'High-assurance' software is an expression used throughout this report that includes the software in systems defined to be 'secure' by the Department of Defense Trusted Computer System Evaluation Criteria as well as other software with very high reliability or security requirements. 'High-assurance' software includes any software which much function as intended or there would be a threat to human life or national security. The report is applicable, then, to high-assurance software which, for the most part, is yet to be developed.

The techniques and technologies investigated by this study are applicable to systems to be certified as 'Beyond Class (A1),' as well as to other high-assurance software. The primary approach in developing this software is formal code verification. This report investigates each Ada construct relative to code verification, the technologies necessary to support code verification, ongoing efforts directly related to verification of Ada code, and alternatives to code verification for achieving high-assurance software.

Since Ada was not developed to be a verifiable language, there are some constructs that will defy formal verification; these challenges do not seem to be overwhelming and could presumably be controlled by restrictions on the use of the language. Tasking and exception handling are the two greatest challenges that the language constructs provide for verification, with tasking being the greater challenge.

The two most critical support technologies for Ada code verification are a formal definition of the language and a formal specification language. The formal definition of Ada developed by DDC needs to be verified, validated, or certified by someone outside of the developing group. This is a major issue. Also, the structure and the syntax of the definition limit its utility. ANNA as a specification language has limitations which are being addressed by Odyssey Research Associates, and alternative forms for specifications are being investigated by Computational Logic.

Two alternatives to formal verification were investigated, the software safety approach and the IBM 'cleanroom.' Each improves the assurance of software yet neither provides adequate assurance to be considered for 'Beyond Class (A1)' software.

Assessing the state of the art of formal verification technology relative to the Ada language requires perspective. To date, the largest code verified system in operation is 4,211 lines of code. Given that languages that are designed and developed to be verifiable provide challenges to the development of large, complex systems, it would be naive to expect that Ada would be easily verifiable.

There are a few general conclusions that have been developed during the course of this study:

Reasons not to use Ada at the Class (A1) level or below are more culturally based than technically based.

Formality in software development should not be all or nothing.

Analysis of constructs that challenge verification can be the basis for developing coding guidelines on software that do not need to be verified.

There are very few people who adequately understand the application of formal methods.

## 1.0 INTRODUCTION AND OVERVIEW

This study had several objectives. The abstract goal was to investigate methods that would lead to a 'high assurance' that software written in the Ada language would perform as intended. To make this goal concrete required a preliminary understanding of 'high assurance.' The Department of Defense Trusted Computer Systems Evaluation Criteria defines secure computer systems as those that satisfy six requirements:

1. There must be an explicit and well-defined security policy enforced by the system.

2. Access control labels must be associated with objects.

3. Individual subjects must be identified.

4. Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party.

5. The computer system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces requirements 1 through 4 above.

6. The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes.

The software in these systems is frequently referenced as 'secure' or 'trusted' software. This report is directly applicable to 'secure' and 'trusted' software and is also applicable to a much broader collection of software. This broader collection includes software with very high reliability or security requirements; software which must function as intended or there will be threat to human life or national security. Throughout this report, such software is referred to as 'high-assurance.'

This study is to review those factors that will affect the use of Ada for high-assurance software. Although formal code verification (i.e., formally establishing the consistency between the software specification and the code) is the primary focus of this paper, other approaches to raising the confidence in software that may be applicable to the security community are investigated.

### 1.1 PROJECT DEFINITION

Initially, several components of the research were identified. One area is to review each Ada construct as defined by the Ada Language Reference Manual (LRM). This review is centered on the impact of each construct on formal verification. The perspective in this component is formal code verification for Ada. The assessment is based on the feasibility to develop a verification axiom, or proof rule, for each construct in isolation. A detailed listing of the constructs and the effect of each on verification is found in Appendix A; a synopsis of this review is contained in Section 2.

After investigating the constructs in isolation, this study considers other factors of formal verification with respect to Ada. This analysis is contained in Section 3.

In addition to the axioms, other support technologies are required for code verification. These include a formal definition and a specification language. Specification languages are not only necessary for formal code verification, but can also be used with other techniques, both formal and less formal. An understanding of runtime issues is necessary to understand the limitations of code verification relative to how the software will function during execution. The status of each of these issues relative to Ada execution is outlined in Section 4.

Of primary interest in this study are two basic questions:

Is the construction of an automated Ada verification environment within the grasp of today's technology?

If yes, what resources would be required to construct the environment?

Section 5 of this report addresses these issues by providing a brief review of related efforts that are ongoing and by projecting, primarily by use of an analogous system, anticipated resources for construction of an Ada verification environment.

Another component of the study is an investigation of alternatives to formal code verification. The two alternatives studied are the verification techniques utilized in the certification of software safety and the human verification of the IBM 'cleanroom' approach developed by Harlan Mills. These approaches are presented in Appendices E and F, respectively.

The remaining sections of this report are structured on the study framework:

Some of the conclusions of this study go beyond the initial study plan. IITRI was chosen to perform this study because of its independence and lack of a vested interest in a particular solution to the challenges of high-assurance software. While investigating the principle questions addressed by the study, secondary observations were made that are included in Section 6.

In addition there are seven appendices. Several topics have been addressed directly by this study which can best be presented by encapsulating the results and placing the information in an appendix. Software safety and the IBM 'cleanroom' each improve the assurance of software, yet neither provides adequate assurance to be considered for 'Beyond Class (A1)' software.

In addition to Software Safety and 'cleanroom' being covered in appendices, automatic programming and Independent Verification and Validation (IV&V) are discussed as alternatives to code verification in an appendix. Automatic programming is covered in response to a belief that if verification technology is not adequately mature to be applied to today's large and complex programs, then perhaps automatic programming is the answer. IV&V is addressed in response to the opinion that if verification is too formal or too complex for the average programmer, then perhaps improved IV&V will adequately increase our confidence in the software that is being produced. Neither is considered a serious alternative to verification: automatic programming because of its own immaturity, and IV&V because of its lack of formality.

Some information that is central to Ada verification is contained in appendices. Background information on research efforts in formal methods applied to Ada is contained in appendices to enhance the flow of this document. This is information on efforts performed at Stanford University and at SofTech.

Each section of this report has a set of assumptions associated with it. The balance of this introduction discusses some of these assumptions and the perspectives against which certain assessments are made.

## 1.2 PRIMITIVE QUESTIONS

### General

The NCSC would eventually like to require formal code verification for certain software efforts. To date, formal code verification has been an expensive, challenging, elusive goal. The objective is to understand exactly what software will and will not do. If code verification is the correct approach, is axiomatic verification the appropriate form of verification?

The current approach for development methodologies is to maximize the use of automation. The benefits of automation are obvious. Tedious, repetitious tasks are performed without human error; hours and hours worth of human computations can be done instantly; rules that can be automated can be mechanically checked with no violations overlooked. However, none of the existing automated systems have been proven to be complete or sound. Is the correct approach to attempt to develop a completely automated system and then to train individuals to use the system?

### Ada-Specific

If Ada code is to be verified, there must be a formal definition that defines precisely what the language means. The Department of Defense (DoD) is diligent about maintaining the Ada standard; who will develop or certify the formal definition? Once a specification language is developed, will the writers of requirements write their requirements in the specification language? Ada is a large, rich language. If it is to be axiomatized, how much should the language be restricted to facilitate axiom development?

### State of Practice

For a system to be certified as Class (A1), a formal model of the security policy must be clearly identified and documented and this model must be shown to be consistent with the formal top-level specification. This consistency must be established by formal methods where verification tools exist. The consistency between the Formal top-level specification and the implementation language is established only informally. Is there any step in this process that precludes use of Ada?

## 1.3 ASSUMPTIONS

In the sections of this report that review Ada constructs (2.0) and the required support technologies (4.0), the assumption is that axiomatic verification of Ada is the intended end result. Appendix A contains a construct by construct review of the literature on the feasibility of axiom development.

The direction of recent Ada verification efforts (Section 5.0) is to perform traditional axiomatic verification. They are assessed under the assumption that to be useful they must facilitate axiomatic code verification.

The assumptions when reviewing alternative approaches (Appendices E and F) change. The assumption here is that software is to be controlled and understood. Software safety attempts to control software; IBM's 'cleanroom' attempts to understand and predict the performance of software. Neither uses axiomatic verification yet both use formal reasoning.

The assumption made in the conclusions (Section 6.0) is that Ada is to be considered for certified software from the Class (C1) through 'Beyond Class (A1).' The focus of this study is Beyond Class (A1), yet some insights on the development of software that is to be certified at a lower level have been gained and are articulated.

## 2.0 ADA CONSTRUCTS THAT AFFECT VERIFICATION

Ada is generally viewed as a rich language. The richness of the language is perceived to be detrimental to formal verification. A construct by construct analysis of the effect of Ada constructs on formal verification is presented in Appendix A. This section highlights the constructs most challenging to verification, identifies a few problems considered to be unresolvable and outlines restrictions on Ada programming style that would be necessary if the code were to be formally verified. The conclusion of this section outlines the impact that using Ada would have on secure systems.

Different perspectives are used in this section. The subsections on challenging constructs, unresolvable problems, and necessary restrictions in coding style assume that the intent is to verify code using axiomatic verification techniques on the code. The concluding section on the impact of the use of Ada for secure systems takes a pragmatic view, assessing the impact of Ada on current practice.

### 2.1 MOST CHALLENGING CONSTRUCTS

Tasks

Far and away, concurrency is one of the most hazardous obstacles in the way of applying formal verification technology to the Ada language. By allowing only restricted use of tasking, it appears that concurrency in Ada can be made amenable to application of formal verification technology; however, it remains to be seen whether what remains has any semblance to what would be called Ada, and whether it would have any usefulness for the objectives for which it was designed into the language. Use of the techniques employed in Communicating Sequential Processors (CSP) [Barringer] appears to be a promising possibility while other research indicates that restriction of communication to only buffers (a la Gypsy) [Young80], to only scalars [Odyssey85], or to only those entry points in which pre-condition and postcondition assertions have been specified [Tripathi] would alleviate many of the inherent difficulties in applying formal verification technology to the Ada concurrency problems.

It must be noted that those restrictions to communicating between tasks reflect the state of the art rather than assess feasibility. Although no one has published proof rules for passing aggregate types, for example arrays or records, development of such proof rules seems quite feasible.

Several researchers [Odyssey85, Pneuli] have recommended that access pointers to tasks not be allowed. The intent of this restriction is to disallow dynamic creation of tasks. In the absence of dynamic creation of tasks, proof rules can be obtained for tasking. However, it is unlikely that this limitation will be readily accepted, particularly in the systems programming arena.

The approach used [Owicki and Gries] to verify Communicating Sequential Processes, Hoare's language framework for concurrent programming, is readily adapted to verification of Ada tasks. This approach consists of two distinct steps: internal verification and external verification. Internal verification consists of proving that the task is an isolated, sequential program. External verification consists of proving that, with the exception of entries, tasks do not affect any subprograms, tasks, or variables declared outside of the task being verified. External verification also requires proof that the task in question is not affected by any subprograms, tasks, or variables declared outside of the task. Again, entries are the exception to this rule. External verification is performed in two states:

a) I-O assertions on entries are made and shared variables are restricted.

b) A proof against deadlocks and starvation is made.

Deadlock and starvation avoidance proofs are prevalent throughout parallel processing literature.

The verification of tasks also assumes the following:

a) All processes terminate normally.

b) Subprogram calls have no side effects.

c) Assignments have no side effects.

d) Tasks may not be aliassed.

Exceptions

The major difficulty with exceptions [Tripathi] in the Ada language from the point of view of verification is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during symbolic execution of programs in the verification step. This complexity is furthered by the fact that exceptions are propagated "as is," which could cause an unhandled exception to propagate from several levels down to a routine that has no understanding of the meaning of the exception. For example, a stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions, the complexity should be reduced. However, the interaction of exceptions and other constructs moves this issue well beyond the problem of bookkeeping. For example, if an exception is raised during execution of a routine with IN OUT parameters, it is not clear if those variables will have been updated prior to transfer of control to the exception handler.

## 2.2 UNRESOLVABLE PROBLEMS

Programming languages not developed for verification inevitably contain constructs that are non-verifiable. Some of these can be controlled through restrictions on programming practices and are discussed in the next section. For two common programming constructs there is no current solution. Although these constructs are not unique to Ada, they do exist in Ada.

Verification of statements including real numbers, and operations on real numbers, is beyond the state of the art. This is due to the lack of accuracy. In the statement

```
J:= (1.0/3.0) + (1.0/3.0) + (1.0/3.0)
```

J would, mathematically, be set to 1.0. However, not only is it uncertain if J will equal 1, it is not known how close to 1 J will be. The effect of this on subsequent statements involving J is unpredictable. As in other languages using real numbers, they cannot be used if the software is to be verified.

Another area that is not verifiable is the process of setting timing constraints. If a section of code must be executed within a specified time, there is no way to verify that the constraint will be met.

## 2.3 RESTRICTIONS TO BE ENFORCED

The recommended coding restrictions of note involve aliasing, aliasing of access types, using shared variables by tasks, and side effects of functions.

Verifying a specific subprogram call requires verifying certain conditions about the parameters involved in the call. These parameters fall into one of two categories: input parameters or output parameters. Input parameters are used only for passing values to the subprogram; output parameters may have their values altered by the subprogram. The conditions that must be verified for each are as follows. No variable, either input or output, may appear in either the precondition or postcondition. No variable that appears in the output parameter list may appear more than once in that list, and no output parameter may appear as an input parameter. The former condition results in updating multiple variables when only one is intended to be updated. For example, if two subprogram formal parameters, A and B, are both passed variable X through a subprogram call, the result of the statements

```
A:=0
B:=1
```

leaves variable X with the value 1 and no variable from the call with value 0. The postcondition after these two statements would assume the existence of two distinct parameters, one with value 1 the other with value 0. If an output parameter appears as an input parameter, the time at which the input parameter is evaluated becomes critical. If the output variable is updated prior to the valuation of the input parameter, the value of the input parameter may differ from the value recorded if the output parameter is not updated prior to evaluation of the input parameter. A single actual parameter used for more than one formal parameter is known as "aliasing."

The association of parameters at subprogram call points would be the ideal location to exclude aliasing [Good80, Odyssey85]. Although there might be a loss of efficiency, the fact that aliasing is unnecessary and complicates application of formal verification technology [Young81] would seem to be sufficient reason for its elimination.

The major concern in the use of access types is the possibility of aliasing (see [Odyssey85] for a lengthy discussion of the matter). One possible solution to the aliasing problem with access types, presented in [Tripathi], is to define a new operator for access types that performs component copying, rather than pointer duplication. This solution is appealing with the advent of the evaluation of the Ada language, due in the latter part of the 1980s, when changes and updates based on several years of working experience with the language will be incorporated into the language. However, restrictions on parameter passing [Odyssey85, Young81] would appear to provide the same benefit with fewer changes.

If a function performs input or output or accesses non-local variables, it is said to cause "side effects." If function subprograms are truly functional they will not include side effects. If Ada functions are restricted to exclude side effects, they can be verified similarly to Gypsy function subprograms, in which these restrictions are enforced by the language.

Shared variables are the major construct in tasking that will have to be restricted (although perhaps simulated through use of other constructs using synchronization) in order to apply formal verification technology to Ada. On this matter, there is no disagreement among the researchers [Cohen, Good80, Odyssey85, Tripathi].

Many of these recommended restrictions are consistent with what are considered good programming practices. The exception is use of shared variables by tasks; forcing tasks to communicate by other means will restrict the utility of tasking. If the code is to be verified, however, the restriction may be necessary.

## 2.4 EFFECT ON SECURE SYSTEMS

Issues are being addressed in this section from various perspectives, and each perspective carries with it some implicit assumptions. The first three subsections have assumed a goal of verifying Ada code and have ignored other issues.

The focus of these three sections is on an axiom system for Ada: which constructs would be most challenging to axiomatize, which will avoid axiomatization, and what restrictions must be made on coding practices to enhance the feasibility of developing an axiom system. Other key elements of verification -- formal semantics, a specification language, and runtime issues -- are addressed in Section 4.0, "Ada-Specific Support Technologies".

This section is a very brief view of current practices to assess the impact of using Ada.

Surprisingly, little is written in the TCSEC about languages, except as they relate to description and to formal specification. For such languages, they are only described as natural languages (e.g., English), or as "formal mathematical languages," with the implication that the latter may be used in formal proofs, possibly automated, demonstrating the consistency between certain entities.

The Class (A1) criteria do not place any requirement, in a formal sense, upon showing that the implementation of a system is logically internally consistent or formally consistent with the Formal Top Level Specification (FTLS). This is the only area in the criteria for the A1 class that makes any specific indication of implementation languages, and the TCSEC discusses its relevance to a system accreditation.

It is instructive to consider the position taken with respect to development of Class (A1) systems implemented in other languages. To date, the only such system is the Honeywell SCOMP. The implementation language used in the SCOMP was C. Restrictions were placed upon the programmers developing code for the SCOMP in order to simplify the process of showing informal correspondence of the implementation with the FTLS. The mapping from an approved design or specification verification tool to Ada, or a slightly restricted version of Ada, would be no more challenging to develop than a mapping from the same tool to C. Since the correspondence could be established for Ada, any decision not to use Ada seems as much cultural as technical.

The TCSEC does define a class of systems that are 'Beyond Class (A1)' in which use of Ada as an implementation language becomes a major factor. At this level, verification is required down to the source code level of the implementation. There is currently insufficient theoretical understanding of Ada to perform such proofs of correctness, and no verification systems to provide automated support of such endeavors.

## 3.0 FORMAL VERIFICATION IN ADA

This section reviews formal verification, the current state of the art, and the current state of practice with respect to Ada.

### 3.1 STATE OF THE ART

Formal verification is the highest technology approach to increasing assurance in the correct functioning of computer software. Other approaches, such as testing, configuration management, or development methodology, have benefits but verification alone can make a quantum leap in the level of assurance. This has resulted in the unfortunate position that verification is an all-or-nothing proposition for software development requiring very high levels of assurance. This mentality has only slowed the application of formal verification technology.

The earmark of formal verification technology, is, of course, the formality of the technology. Unfortunately, this formality is a wicked two edged sword. Great benefit accrues in being able to provide a functional description (a model) of the intended behavior of a program, and to be able to provide a mathematical proof of consistency between that model and the software implementation of it. However, to take advantage of these benefits, it is necessary to ensure that the underlying mechanisms, proof techniques, and automated support tools are themselves sufficiently trusted not to introduce errors in one manner or another. The development of such constituent elements is also not a mild undertaking, requiring individuals with advanced education in the appropriate field. Furthermore, the use and application of the technology also requires individuals with similarly advanced training.

The formal verification process consists of preparing, prior to the development of software, the formal specification of a model of the intended behavior of the software. Some effort may be placed (as described previously) in the analysis of the specifications to ascertain their completeness and internal consistency. Then, following the software development methodology, designs and implementations at the various levels of the software are completed, and formal correspondence with the specification is performed, resulting in proofs of correctness of the implementation with respect to the specification. The formal proof of correctness consists of showing that the two separate, hopefully somewhat orthogonal, descriptions have a proper correspondence.

Besides the level of expertise required by the individuals performing the formal specification and verification, the application of this advanced technology requires such significant amounts of available computing resources as to dwarf other methods of increasing assurance. Without sufficient computing resources being brought to bear on the development, it would be no wonder that the technology could be faulted for the failure of such projects attempting to utilize formal verification technology. The use and application of this technology will also greatly increase the difficulty of project management. It has already been difficult enough to manage large software projects in the absence of formal verification technology. With such application, there is expected to be at least an additional equivalent amount of specification as implementation, and supporting proofs, whose size has been conservatively estimated at ten times the size of the specification and implementation combined.

## 3.2. ANALYSIS OF THE TECHNOLOGY WITH RESPECT TO ADA

As mentioned above, one of the earmarks of formal verification technology is its formality. This is an area that seems to have had varied amounts of support during the design of Ada. In the early requirements documents for the Ada language, verification was mentioned as a desirable goal, but the language contains many constructs (see Appendix A) that prevent this goal. In order to do formal proofs of consistency between the specification and the implementation, a formal description of the language semantics is also necessary. Some effort has been done by the EEC in this area, but it has not sufficiently matured to a stage where it can be utilized in formal verification. Even the only viable specification language for Ada, ANNA, has been geared more toward the utilization of runtime assertion checks, not formal verification, and has largely ignored the aspects of parallelism. At least one known effort is involved in extending ANNA to overcome these deficiencies.

Once the remaining theoretical obstacles have been overcome, it will be necessary to develop automated support tools for the specification and verification process.

## 4.0 ADA-SPECIFIC SUPPORT TECHNOLOGIES

Formal code verification requires several key components. The implementation language must have a formal definition or semantics so that the exact meaning of each language construct and sub-construct is clear and unambiguous. There must be a specification language. Since the proof establishes the consistency between the specification and the code, the specification must be stated in a formal language. Although not required for code verification, runtime issues are very important. Code, even if proven correct, will not function as expected if the runtime environment does not execute in a manner that is consistent with the assumptions of the proof.

These issues -- formal definitions and semantics, specification languages, and runtime issues -- are discussed in this section.

### 4.1 FORMAL DEFINITION AND FORMAL SEMANTICS

The most ambitious attempt at a formal definition of the Ada language is being undertaken by the Dansk Datamatik Center and its member companies. This definition is intended to give meaning to each Ada language construct by providing meaning to each sub-construct. This definition is to be a readable, unambiguous definition that will be implementation dependent. The approach was to develop a static semantics and then to develop the dynamic semantics. The dynamic semantics will have embedded in it the sequential constructs, as these may be executing in parallel, and the input-output portions of the language.

The semantics are provided by use of axioms which are given as abstract data types and an algebra, or model, for combining the constructs. In addition to being a basis for formal proofs, this definition is meant to be a standard reference or specification for implementers of the language.

This effort is producing a very large volume for the formal definition. Although the developers are building into the definition mechanisms to establish the completeness and consistency of the definition, these two concerns -- consistency and completeness -- are still major.

SofTech has been working on an effort to define the problems and potential solutions to the development of an axiomatic semantic definition of the Ada language. The difference between an axiomatic semantic description of Ada and the definition of Ada given by MIL-STD-1815A is that the semantic description defines the behavior and interrelationships of the individual language constructs in such a way as to be used as the basis of a proof. The existence of a semantic definition of a language is necessary if a comprehensive verification technology is to be developed for that language. Any aspects of a language that are not rigidly, semantically defined are subject to varying interpretations by different compilers. Some of the Ada constructs that pose difficulties in verification have been left out of the semantic description. A list of the excluded constructs are address clauses, unchecked conversions, variables shared among tasks and subprogram calls that generate aliases.

To a large extent, most elements of a semantic description are handled at compilation time and need not be dealt with during verification time. It is important to realize that the actual verification environment is based on the semantic definition of Ada rather than the actual language, and constructs that are not included in the semantic definition invalidate the verification process. The SofTech study concerns itself only with those constructs that are not dealt with at compilation time. This definition is described in detail in Appendix C.

SofTech has developed the architecture for a verification environment based upon the formal semantic definition they have developed for Ada. This verification environment is based upon a modification of Ada -- the semantic definition -- and is described in detail in Appendix D.

## 4.2 SPECIFICATION LANGUAGES

Specification languages are necessary for code verification and can also be used for other proof-related purposes. Analysis of the specifications prior to proving consistency between the code and the specification, can only be formally done with a mathematically-based specification language. Also, runtime analysis is facilitated by use of a specification language to state the assertions that are to be checked at runtime. Some Ada-specific work on specification language tools is being done at Stanford University and is discussed briefly in Section 5.1 and in more detail in Appendix B. Use of specifications in runtime is discussed in Section 4.2.

At the present time, software system specifications are done in the English language. While using English as a specification language has the advantage of providing easily readable, easily composed specifications there are some problems inherent with the use of English. The English language often contains inconsistencies and ambiguities which inhibit exact interpretations of the specifications. The translation required from specification language to coding is so broad due to the vast difference in media as to create transitional errors in all but the most detailed, trivial or exhaustively used system.

The principle specification language for Ada was developed at Stanford University. ANNA (ANNotated Ada) is an annotation language for all constructs of Ada except tasking. The language is designed to support various theories of formally specifying and verifying programs. One area of current research is the use of parallel processors to provide concurrent checking of specifications.

Since the ANNA semantics closely parallel those of Ada, its use in secure systems development would allow the system designers and implementors to use the same underlying language semantics for communication of the intended behavior of their specifications and programs. However, since the language appears to have been targeted to the runtime validation of program execution rather than pre-execution proofs of correctness, its applicability in secure systems development would be limited until a supporting infrastructure, both in terms of theoretical aspects of the language and in terms of automated tools, can be developed.

Norm Cohen has worked on the development of Ada as a specification language at Softech. Use of a formal language, such as Ada, to describe such specifications would greatly reduce the number of translational errors. This could be used as an important first step in developing trusted verifiable software.

The use of Ada as a specification language enables the specification to be read and interpreted by a compiler-like consistency checker which is able to enforce internal consistency within the semantics. Taken to a higher level, the consistency checker may be used to check the consistency between different levels of specification. In this manner the integrity of the initial specification may be checked, level by level, down to the actual code. If the compiler being used is a trusted compiler then the consistency of all specification level transitions have been verified from the most abstract level to the actual code.

## 4.3 RUNTIME ISSUES

Verification of a program, in any language, takes place during a "proof time" which occurs before the program is executed. Situations that are difficult to predict at proof time are generally either discounted or disallowed by verification techniques. The result of this is that one of two things happens: either an issue is ignored or discounted in some superfluous way, or a great deal of effort is spent attempting to suppress possible occurrences of the problem.

Ada deals with runtime difficulties through the use of exceptions. Much work has gone into exception handling during verification. Two of the more detailed verification systems are Cohen's work at SofTech, which is described in Appendix C, and McHugh's work at the University of Texas at Austin in the Gypsy language [McHugh]. Gypsy's exceptions are similar to Ada; this enables us to apply runtime techniques developed in Gypsy to Ada.

McHugh handles exceptions in two manners, one of which is that exceptions that are considered domain related. These exceptions are discounted with regards to verification. An illustration of this is as follows: a verified satellite communications system would fail if the satellite were disabled; or on a more local basis, a trusted operating system may give faulty retrievals if the memory is damaged. The effect of this decision is to localize the responsibility of the verification to not include errors that emit from outside the program. Should the satellite or memory be verified in addition to the software, then a memory failure would indicate a fault in the verification process.

Exceptions that are not external in origin are handled differently. These exceptions are, in effect, eliminated from the program to be verified. Exceptions of this type are indirectly optimized during the verification process before runtime. This is performed by the creation of optimization conditions that are related to possible exceptions. Optimization conditions must be sufficiently well defined to show that the corresponding optimization condition must occur before the exception may be raised. Given this, it is easily proven that, if an optimization code can be proven to never occur, the exception will never be raised. It is easily concluded that an exception which is never raised cannot compromise the verification of a program or module of a program. We may now state that, if an optimization condition is offered as a precondition of a module of Ada code then the code may be considered verified with respect to the exception that corresponds to the optimization condition.

In short, the nature of exceptions makes them difficult to verify in a robust manner. The state of the art is little more than the statement that "if an exception never occurs it creates no problems in verification."

## Runtime Assertion Checks

Another area in which specifications may be applied is that of runtime assertion checks. This section describes the various types of such checks, their applicability and utility, and the status of the technology as applied particularly to Ada.

Runtime assertion checks can increase the assurance in the correct functioning of a program in a number of ways. First, the additional effort expended in the development of such assertions, whether they be informal or formal, increases the level of the programmer's understanding of the program. Second, the preparation of assertions can provide a gentle introduction to the application of formal verification technology (see below), by allowing the programmer to get a small amount of exposure to part of the verification process without having to make the total investment in learning the process at once. Finally, and the major reason for their use, is that the runtime checks can be used in instrumented versions of the executable programs to check the programmer's understanding of the program against its actual execution.

Runtime assertion checks can be included in a program in various forms. The first, and most obvious, form of assertion is simple inclusion of code in the programming language itself. This code may be instrumented in such a way as to be turned on or off at runtime, although recompilation of the source code may be required. The level of overhead associated with such checking increases from the lowest, in which the runtime assertions are not included at compile time, followed by instrumentation with checks turned off, runtime assertions compiled in, to instrumentation with checks turned on. Another possible form of inclusion of runtime assertions is through the use of a formal assertion mechanism. These may be processed by a preprocessor, as in the case of the C language assert construct, and converted into corresponding source code, or parsed with the program text, as in ANNA runtime specifications, and expanded during the code generation phase of compilation.

One benefit of the use of runtime assertion checks is that the technology is so similar to compiler technology that it can be applied without additional technology development. Its application is also at a sufficiently low level to allow its use by programmers at various ability levels (depending upon the level of formality of the specification language). Another benefit is that the technology can be easily integrated into the traditional software development methodology without having to make large investments in retraining, changes in practice, or additional hardware.

The use of runtime assertion checks is not without drawbacks, however. The most obvious one is the overhead penalty in execution time while running programs instrumented with such runtime assertions. Another drawback is that the application of formal verification technology may obviate such runtime checks. With a formal verification methodology, it may be possible to logically prove that the assertion holds at the point in the program's execution, and thus the resulting runtime check can be omitted, thus reducing the program's runtime overhead and increasing its performance.

Runtime assertion checking is a technology which can be, and currently is being, applied to increase the assurance in the correct execution of software written in Ada. Research at Stanford University has resulted in ANNA, a specification language for Ada, specifically designed for use in preparing runtime assertion checks. Automated tools for supporting the software development process using such checks have been developed, and preliminary results have been obtained on a number of research and development projects.

ANNA was designed primarily for use with the sequential aspects of the Ada language. Efforts are underway to extend ANNA and combine it with other languages to use it for the parallel aspects as well. Additional research is being targeted at providing a mechanism for concurrent execution of the resulting runtime assertion checks (on multi-processor hardware) in order to exploit some of the benefits of the parallel execution and reduce the apparent runtime overhead penalty associated with the checks.

The use of ANNA is very CPU intensive, not only in the execution time of the resulting software, but also in the execution time of the automated tools. The lack of speed in these tools is due in part to their use of somewhat dated compiler technology, as well as the fact that they are implemented in Ada, where compiler maturity is not at a sufficient level to produce quality code comparable to other languages. This drawback might prevent its application in environments that are unable to provide a sufficient hardware base for development environments, although the benefit of the choice of Ada has allowed transition among available hardware configurations which provide Ada software development environments.

## 5.0 SCOPE OF AN ADA VERIFICATION ENVIRONMENT

The NCSC is interested in pushing the state of the art in certification to 'Beyond Class (A1).' This translates into formal code verification. The DoD is insistent on the use of the Ada programming language. Therefore, formal code verification of Ada software is a goal of the Center. Due to the nature of formal code verification, this process should be highly automated in an Ada verification environment. This section of the report briefly describes three ongoing research efforts in this direction and then discusses potential necessary resources to develop a functional Ada verification environment.

### 5.1 RECENT EFFORTS

There are currently three known efforts investigating various aspects of this research. These are efforts being undertaken by Stanford University, Computational Logic, Inc., and Odyssey Research Associates Inc.

The Stanford University research has focused mostly on the development of a specification language, ANNA, and the development of tools to support its use in the area of runtime assertion checks. The language was completed several years ago, and is being used by a number of contractors and other researchers, both in real world applications and in further research. The automated support tools themselves are written in Ada, and run on a number of different hardware architectures. Neither the language, nor the support tools have been designed specifically for use in a verification environment, although much of the effort that has been completed would provide a starting basis for such an environment.

The current effort is part of a three year project (completion September 1989) to construct a prototype environment of tools for software and hardware development. These tools are based on specification languages with particular emphasis on distributed computing and implemented in the Ada language for maximum portability to various environments. It is hoped that the results of this effort will provide a better understanding into what features are necessary in a development environment with a number of possible applications: requirements analysis and negotiation, rapid prototyping, formal implementation guides, automatic implementations from specifications and construction of self-testing systems.

The effort draws on a significant amount of already completed research, particularly that done in developing the specification languages ANNA and TSL, and the effort already performed in developing tools for syntactically parsing the ANNA text and manipulation of the underlying DIANA representation.

The emphasis appears to be similar to that being taken by other researchers, in attempting to apply specification and verification techniques over the entire spectrum of system development, from the top level system requirements and definition down to the low level hardware implementation.

In addition to the primary emphasis on the development of a prototype environment for software and hardware development, additional emphasis is being placed on developing reusable components that may be shared among the various applications level programs in the project. The use of Ada as the implementation language is intended to aid in the portability of the resulting system, and the emphasis on integrability (the commonality of tool interfaces, underlying structures, etc.) will also allow the investigation of the utility of such an approach in a large development project. The choice of Ada is also intended to allow the investigation of features of parallelism in specification, runtime checking and software development that have heretofore been unavailable in other environments and with other languages.

Computational Logic, Inc. is investigating the underlying logic necessary to support an Ada verification environment. Leveraging their background with the Gypsy Verification Environment, the Boyer-Moore theorem prover, and the beginning research on the underlying logic for the Rose language (a Gypsy successor), various constructs in the Ada language are to be analyzed to determine the underlying logic structure necessary to support such a verification environment. This effort is more directed at preparing the underlying formalisms, and understanding the relationship between those formalisms and the programming language semantics than to the development of any additional specification languages or automated support tools.

Odyssey Research Associates, Inc. is involved in research and development of a prototype Ada verification environment. The major thrust of the effort is the extension of ANNA to support real-world programs. ANNA will be extended by applying it to known examples; finding and fixing the shortcomings in each of these applications. Additional effort is planned to design and develop a prototype verification environment supporting the ANNA extensions with a theorem prover and proof rules and formal semantics for that portion of the Ada language being utilized.

As these various approaches to formal verification with the Ada language are progressing, it will be necessary to apply the resulting technology in order to gain experience with it and to evaluate the feasibility for development of large scale projects.

## 5.2 PROJECTIONS OF RESOURCES

Software cost estimating is far from an exact science. The primary input to cost models is lines of code and secondary inputs include program application type, complexity of the program, capability of the developers and other environmental factors. Even if these factors were known, attempting to use a traditional parametric cost model for estimating the cost of an Ada verification environment would be impractical. The major issues in building the environment require answering several questions that are in the research stage.

The primary concern is concurrency. Proof rules for Ada tasking are still a research issue. Also, the utility of the DDC formal definition of the language, or some alternative formal definition, is not yet established. There is no Ada specification language that can adequately express the logic required to be the basis of a proof system.

In spite of these qualifications, an estimate will be made. The process used to derive the estimate will be analogy, where a system with some similarity to the desired system will be used as a basis for the estimate. The system to be used as the analog is the Pascal verifier developed in the 1970s at Stanford University.

Development of the Pascal verifier took approximately 20 man years of effort during the years 1972 to 1979. This system did not contain all of the elements that would be desirable in a system to be developed today. Those elements would include a friendly user interface, counter-example generators, reusable proofs, proof classifiers and tracing capabilities to assist in following the progress of the proof. Also, the system was not built using the same standards of software engineering that would be used today, which may increase development effort in exchange for decreasing maintenance effort.

The estimate for developing a verification environment for sequential Ada that had the following capabilities in 30 man years of effort:

Prove the consistency between a specification and the behavior of a program,

Perform runtime checking of consistency of specifications, and

Perform analysis of specifications.

To develop an environment that would include tasking, with clearly articulated and enforceable restrictions, would take much more effort. Some estimate of that could be made based on the success of the Odyssey Research Associates development of a prototype environment that will handle a 'cluster' of Ada with some restrictions.

The Odyssey effort, believed to be approximately 30 man years, is attempting to develop an environment that supports the Ada language to the extent that the Gypsy Verification Environment supports the Gypsy language. If the prototype is successful, the development of a production quality environment would require, presumably, an additional 30 man years.

The estimates are that an environment for sequential Ada could be built for 30 man years; if the Odyssey effort is successful and the technology developed by them is accessible, then enhancing that effort to production quality would take approximately an additional 30 man years. These estimates are obviously very rough.

## 6.0 CONCLUSIONS

This study had several objectives. The abstract goal was to investigate methods that would lead to the highest assurance that software written in the Ada language would perform as intended. This led to the examination of elements related to the formal verification of Ada software, to the examination of formal methods applied at levels other than code verification, and to the examination of less-formal methods.

Relative to code verification, the continuing examination of Ada constructs reveals two findings. Since Ada was not developed to be a verifiable language, there are some constructs that will defy formal verification; these challenges do not seem to be overwhelming and could presumably be controlled by restrictions to the use of the language. Tasking and exception handling are the two greatest challenges that the language constructs provide for verification. Of these, tasking is the far greater challenge.

Code verification requires both a formal definition and a specification language. The formal definition being developed by DDC will need to be verified, validated or certified by someone outside of the developing group. This is a major issue. Also, the structure and syntax of the definition will limit its utility.

ANNA as a specification language has limitations which are being addressed by Odyssey Research, and alternative forms for specifications are being investigated by Computational Logic.

As these various elements of formal verification with the Ada language progress, it will remain to apply resulting technology in order to gain experience with it and to evaluate the feasibility for development of large scale projects. To date, applications of verification technology have been performed by small groups of people on small tasks, with rather limited results in terms of both costs and quantity of software. This situation will be no different in application of formal verification technology to the Ada language. The community of individuals trained in the use and application of formal verification techniques is small, and the intersection of those individuals with the limited pool of talent proficient in the Ada language continues to reduce the available labor.

Existing verification projects have been small in size, because that has been the only manner in which to maintain control of the complexity of the project, and to be able to support the project with automated support tools. Advances in hardware technology will improve the utility of support tools, but cannot solve the personnel problem.

Beyond code verification, formal methods are being investigated with respect to Ada. The two areas of research are the application of formal methods to specification analysis and to runtime assertions. In the area of specification analysis, the focus is on finding and analyzing inconsistencies in the specifications. Although this will not provide the assurances of code verification, it seems that an emphasis on this work will prove to support code verification in the long run, and will be useful in its own right. Successful verification projects depend on consistent specifications.

Although runtime assertion checking seems redundant for verified software, this avenue seems particularly worthy of research for distributed systems. This avenue may well help in the understanding of concurrency.

Two alternatives to formal verification were investigated, the software safety approach, and the IBM 'cleanroom.' Each improves the assurance of software, yet neither provides adequate assurance to be considered for 'Beyond Class (A1)' software.

Assessing the adequacy of the state of the art of formal verification technology relative to the Ada language requires perspective. To date, the largest code verified system in operation is 4,211 lines of code. Given that languages that are designed and developed to be verifiable provide challenges to the development of large, complex systems, it would be naive to expect that Ada would be easily verifiable.

There are a few general conclusions that have been developed during the course of this study:

Reasons not to use Ada at the Class (A1) level or below are more culturally based than technically based.

Formality in software development should not be all or nothing.

Analysis of constructs that challenge verification can be a basis for developing coding guidelines on software that does not need to be verified.

There are very few people who adequately understand the application of formal methods.

## 7.0 REFERENCES

Ada Programming Language, ANSI-MIL-STD-1815A, Department of Defense, 22 January 1983.

Barringer, H., Mearns, I., "Axioms and Proof Rules for Ada Tasks," IEEE Proceedings, Volume 29(E), Number 2, pp. 38-46, March 1982.

Berg, H.K., Boebert, W.E., Franta, W.R., Moher, T.G., Formal Methods of Program Verification and Specification, Prentice-Hall, Inc., Englewood Cliffs, NY 07632, 1982.

Cohen, Norman H., "Ada Axiomatic Semantics: Problems and Solutions," SofTech, Inc., One Sentry Parkway, Suite 6000, Blue Bell, PA 19422-2310, May 1986.

Dahl, Ole-Johan, "Can Program Proving Be Made Practical?," Institute of Informatics, University of Oslo, Norway, 1978.

DiVito, Ben, "A Verifiable Subset of Ada," TRW, unpublished manuscript.

"The Draft Formal Definition of ANSI-MIL-STD 1815A Ada," EEC Multi-annual Programme, Project No. 782, Annex 1, Version 14-12-1984, Dansk Datamatik Center, Lundtoftevej 1C, DK-2800 Lyngby, Denmark.

Ernst, G.W., and Hookway, R.J., "Specification and Verification of Generic Program Units in Ada," Department of Computer Engineering and Science, Case Institute of Technology, Case Western University, Cleveland, Ohio.

Floyd, R.W., "Assigning Meanings to Programs", Proceedings of the American Mathematical Society Symposia in Applied Mathematics, Vol. 19, pp. 19-31.

Gerhart, Susan L., "Fundamental Concepts of Program Verification", AFFIRM Memo-15-SLG, University of Southern California Information Science Institute, Marine Del Rey, CA 90291, February 18, 1980.

Gerth, R. "A Sound and Complete Hoare Axiomatization of the Ada Rendezvous", Proceedings of the 9th International Coloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 140, Springer Verlag, pp. 252-264, 1982.

Gerth, R. and deRoever, W.P., "A Proof System for Concurrent Ada Programs", RUU-CS-83-2, Rijksuniversiteit Ultrecht, January 1983.

Good, Donald I. et al., "Report on the Language Gypsy", Version 2.0 The University of Texas at Austin, ICSCA-CMP-10, September 1978.

Good, Donald I., Cohen, Richard M., and Keeton-Williams, James, "Principles of Proving Concurrent Programs in Gypsy", Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX 78712, January 1979.

Good, Donald I. et al., "An Evaluation of the Verifiability of Ada", September 1980.

Goodenough, John B., "Exception Handling: Issues and a Proposed Notation", Communications of the ACM, 18(12):683-696, December 1975.

Guttag, John V., Horowitz, Ellis, and Musser, David R., "Abstract Data Types and Software Validation", Communications of the ACM, 21(12), December 1978.

Hantler, Sidney L., and King, James C., "An Introduction to Proving the Correctness of Programs", Computing Surveys, Vol. 8, No. 3, pp. 331-353.

Hill, A.D., "Asphodel - An Ada Compatible Specification and Design Language", (unpublished manuscript), Central Electricity Generating Board, Computing and Information Systems Department, Laud House, 20 Newgate Street, London EC1A 7AX, U.K.

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Communications of the ACM, 12(10):576-581, October 1969.

Hoare, C.A.R., and Wirth, Niklaus, "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica, :2, 19876.

Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., Wichmann, B.A., "Preliminary Ada Reference Manual" and "Rationale for the Design of the Ada Programming Language", ACM SIGPLAN Notices, 14(6), June 1979.

"Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada", HQ85-29920-1, Institute for Defense Analyses, 1801 N. Beauregard St., Alexandria, VA 22311, May, 1985.

Kemmerer, Richard A., Verification Assessment Study, Final Report, National Computer Security Center, C3-CR01-86, Ft. George G. Meade, MD 20755-6700, March 27, 1986.

Formal Definition of the Ada Programming Language, Institute National de Recherche en Informatique et en Automatique, November, 1980.

Liskov, Barbara H., and Snyder, "Alan, Exception Handling in CLU", IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, November 1979.

Luckham, David C. and Suzuki, Norihisa, "Verification of Array, Record, and Pointer Operations in Pascal", ACM Transactions on Programming Languages and Systems, pp. 226-244, October 1979.

Luckham, David C. and Polak, Wolfgang, "Ada Exception Handling: An Axiomatic Approach", ACM Transactions on Programming Languages and Systems, 2(2):225-233, April 1980.

Luckham, David C., von Henke, Friendrich W., Krieg-Brueckner, Bernd, Owe, Olaf, "ANNA -A Language for Annotating Ada Programs, Preliminary Reference Manual", Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

McGettrick, Andrew D., Program Verification Using Ada, Cambridge Computer Science Texts - 13, Cambridge University Press, 1983.

McHugh, John, "Towards Efficient Code from Verified Programs", Technical Report ICSCA-40, Institute for Computing Science, University of Texas at Austin, March 1984.

Mills, Harlan, "Human Verification in Ada", Presented at the Third IDA Workshop on Ada Specification and Verification, Research Triangle Park, NC. May 14-16, 1986.

Nyberg, Karl A., Hook, Audrey A., Kramer, Jack F. "The Status of Verification Technology for the Ada Language", Institute for Defense Analyses Paper #P-1859, IDA, 1801 N. Beuregard St., Alexandria, VA 22311, July 1985.

Odyssey Research Associates, Inc., "A Verifiable Subset of Ada", (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850, September 14, 1984.

Odyssey Research Associates, Inc., "Toward Ada Verification", Preliminary Report (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, March 25, 1985.

Owicki, S.S., and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach", Communications of the ACM, Vol. 19, No. 5, (May 1976), 279-285.

O'Donnell, Michael J. "A Critique of the Foundations of Hoare Style Programming Logics", Communications of the ACM, Vol. 25, No 12 (December 1982), 927-935.

Pneuli, A., and deRoever, W.P., "Rendezvous with Ada - A Proof Theoretical View", Proceedings of the AdaTEC Conference on Ada, Arlington, VA, pp 129-137, October 1982.

Shaw, Mary, "The Impact of Abstraction Concerns on Modern Programming Languages", Carnegie-Mellon University Computer Science Department, Technical Report CMU-CS-80-116, Pittsburgh, PA, April 1980.

Sutherland, David, "Formal Verification of Mathematical Software", NASA Contract Report 172407, Odyssey Research Associates, 1984.

Tripathi, Anand R., Young, William D., Good, Donald I., "A Preliminary Evaluation of Verifiability in Ada", Procedings of the ACM National Conference, Nashville, TN, October 1980.

Young, William D., Good, Donald I., "Generics and Verification in Ada", Proceedings of the ACM Symposium on the Ada Language, Boston, MA, pp. 123-127, 9-11 December 1980.

Young, William D., Good, Donald I., "Steelman and the Verifiability of (Preliminary) Ada", ACM SIGPLAN Notices, 16(2):113-119, February 1981.

## APPENDIX A

## AN OVERVIEW OF FORMAL VERIFICATION TECHNOLOGY APPLIED TO THE ADA* PROGRAMMING LANGUAGE

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

## 0. Preliminary

This Appendix reviews the state of the art of program verification with respect to the Ada language. Where relevant, references are made to existing verification languages and systems that implement features equivalent to Ada, as well as those with only limited correspondence. Special note will be made of programming constructs that pose difficulty to verification in general, and Ada constructs in particular that pose such difficulty, both in isolation, and when taken in concert with other constructs.

The perspective on formal verification of this Appendix is that of Hoare and Floyd. Briefly, each statement is preceded by an assertion (the precondition) and followed by an assertion (the postcondition). These assertions are statements that involve program variables. A proof consists of demonstrating two things. The first is to show that if the precondition is true, and the statement executes, then the postcondition is true. This requires the existence of a proof rule for each type of statement to be executed. The proof rule defines how the statement alters the precondition to produce the postcondition. The second part of the proof requires demonstrating that the postcondition of a statement logically implies the precondition for the subsequent statement.

One of the major deficiencies of the Ada language with respect to verification is the lack of a formal definition of the semantics of the language. Although the Ada Language Reference Manual (LRM) may be considered to provide somewhat of an "operational" semantics, it is not sufficiently formal to be applied in the use of formal verification technology. The need for a formal, semantic definition is based on the need to specify which of the phrases in a syntactically correct program are commands, and what conditions must be imposed on an interpretation in the neighborhood of each command [Floyd]. Each proof rule must be validated by interpreting it with respect to the formal definition of the language to which it is to be applied. Also, this formal definition provides a final authority in disagreements; transforms the system of reasoning into a mathematical object; and enables the processing of the system to be automated [O'Donnell]. An effort is currently underway to provide a formal definition [EEC, LNRC] that will hopefully provide the mathematical foundation of language semantics. Once this foundation has been laid, attention can then be turned to the development of proof rules for the constructs in the language, which are vital for the application of formal verification technology.

In addition to a formal semantics, Ada will require a specification language if formal verification of Ada programs is to be achieved. A specification language provides the vehicle for stating, in mathematically precise terms, what a program is expected to do. ANNA is a specification language for Ada and it, or an extension of it, may prove satisfactory for verification purposes.

The remaining chapters of this Appendix are structured to be in accordance with the chapters of the LRM. Each section describes the state of the art of formal verification technology with respect to the general topic (as outlined in the LRM); the state of the art of formal verification technology in particular as it relates to the given Ada construct; difficulties in applying formal verification technology with that construct; and possible solutions to allow the application of formal verification technology through certain restrictions on the use of the construct, whether alone, or in concert with other constructs of the language. Many of these recommendations are drawn from the existing literature [Cohen, Divito84, Odyssey85, Tripathi, Young80, Young81] specifically addressed toward making Ada a language in which formal verification technology can be applied.

# 1. Introduction

For the most part, this chapter of the LRM deals with matters that are unrelated to the syntax and semantics of the Ada language, and, as such, is irrelevant to the application of formal verification technology. One area that does have some relevance, though, is the area of "Classification of Errors" (1.6).

## 1.1 Scope of the Standard

No further implications for formal verification technology.

### 1.1.1 Extent of the Standard

No further implications for formal verification technology.

### 1.1.2 Conformity of an Implementation with the Standard

No further implications for formal verification technology.

## 1.2 Structure of the Standard

No further implications for formal verification technology

## 1.3 Design Goals and Sources

No further implications for formal verification technology

## 1.4 Language Summary

No further implications for formal verification technology.

## 1.5 Method of Description and Syntax Notion

No further implications for formal verification technology.

## 1.6 Classification of Errors

The LRM defines four types of errors: compilation time errors, runtime errors, erroneous execution, and incorrect order dependencies. The first of these four types of errors, compilation time errors, are commonly referred to as syntax errors, and will not allow for compilation of the program. Since one of the requirements of formal verification is that the code to be verified be "legal" with respect to the rules of the language, this type of error will preclude verification. The second type of error, the runtime error, occurs during the attempted execution of the program, and is commonly referred to as an "exception." Runtime errors in general have been widely discussed in the literature [Goodenough, Liskov], and have even been discussed with respect to Ada [Luckham80]. In addition to this, since Chapter 7, "Exceptions", is devoted entirely to this topic, its discussion will be deferred to that chapter. The remaining error types, erroneous execution and incorrect order dependencies, are not required to be detected at compilation or execution, but do result in violations of certain rules of the Ada language. Since the detection of these errors is not required by the LRM, their detection falls to the verification environment.

This chapter of the LRM deals with the delineation of the elements which make up the parts of the language. These elements define the "tokens" which are processed in determination of the legality of a program. The majority of these items has little relevance to the issue of formal verification technology. Comments are noteworthy, in that the ANNA specification language [Luckham84] defines "formal" comments meant to be used in the specification of Ada software. These comments are not executable, and therefore will not influence the execution of the software, however, they will influence the verifier, and as such will influence the verification process.

## 2.1 Character Set

No further implications for formal verification technology.

## 2.2 Lexical Elements, Separators, and Delimiters

No further implications for formal verification technology.

## 2.3 Identifiers

No further implications for formal verification technology.

## 2.4 Numeric Literals

No further implications for formal verification technology.

### 2.4.1 Decimal Literals

No further implications for formal verification technology.

#### 2.4.2 Based Literals

No further implications for formal verification technology.

### 2.5 Character Literals

No further implications for formal verification technology.

### 2.6 String Literals

No further implications for formal verification technology.

### 2.7 Comments

No further implications for formal verification technology.

### 2.8 Pragmas

The **PRAGMA** construct is relevant, although this chapter of the LRM does not discuss its use and application, but only rules for its placement within the program text. For the discussion of its interaction with formal verification technology, see Section 11.7.

### 2.9 Reserved Words

No further implications for formal verification technology.

### 2.10 Allowable Replacement of Characters

No further implications for formal verification technology.

## 3. Declarations and Types

This chapter of the LRM defines the type mechanism, the means for declaring objects of the types, and the set of operations on the types. The major areas which are of concern to the application of formal verification technology include object declarations, real types, and access types.

### 3.1 Declarations

No further implications for formal verification technology.

### 3.2 Objects and Named Numbers

#### 3.2.1 Object Declarations

One manner in which erroneous programs may occur is in the use of an object prior to assigning a value to the object. In other languages designed for verifiability (e.g., Gypsy), the formal definition of the language requires that the default value for objects be specified when declared [Good78]. When the semantic definition of Ada is complete, it should resolve this difficulty. Another approach to this problem prior to the completion of the semantic definition is to disallow references to objects before their initialization [Odyssey85]. This is addressed through explicit initialization, and hopefully enforced by automated verification tools when performing symbolic evaluation and path analysis. However, even these rather laborious steps can not fully assure that a variable is defined before it is referenced.

### 3.2.2 Number Declarations

No further implications for formal verification technology.

## 3.3 Types and Subtypes

For the most part, types and subtypes within Ada are well behaved with respect to formal verification. With the exception of arrays and access types, verification is restricted to constraint and accuracy difficulties. Verification techniques intent on prevention of constraint errors are well documented [Good78, Hantler, Hoare69, McGettrick]. Nondiscrete types, such as real and floating point types, require verification techniques to prevent inaccurate values. Methods of measuring possible inaccuracies, short of reporting the maximum possible error, are not available. The existence of a default value for declarations is used to verify that no undefined variable will be referenced.

### 3.3.1 Type Declarations

No further implications for formal verification technology.

### 3.3.2 Subtype Declarations

No further implications for formal verification technology.

### 3.3.3 Classification of Operations

No further implications for formal verification technology.

## 3.4 Derived Types

Derived types are subject to the same restrictions as their parent types.

## 3.5 Scalar Types

No further implications for formal verification technology.

### 3.5.1 Enumeration Types

No further implications for formal verification technology.

### 3.5.2 Character Types

No further implications for formal verification technology.

### 3.5.3 Boolean Types

No further implications for formal verification technology.

### 3.5.4 Integer Types

No further implications for formal verification technology.

### 3.5.5 Operations of Discrete Types

Discrete types require several restrictions to verification procedures. Verification techniques must be used to prevent constraint errors such that OP (M,N) remains within the bounds of T if M and N are of type T, and OP is any operation valid on T. Some mathematical laws are not applicable on discrete types, in particular the associative and distributive laws regarding statements are not applicable due to possible constraint errors.

### 3.5.6 Real Types

Real types have been largely ignored in formal verification technology. Part of this difficulty has been due to the fact that operations on such types decrease the accuracy of the result, as a function of the values and the underlying implementation [McGettrick, Tripathi, Young81], thus precluding the proofs of correctness. Proofs of correctness would require precise measurement of the accuracy. This measurement is unavailable as the exact representation of a real within the machine is not defined (e.g., a machine capable of representing four digits may represent one half as either 0.500E00 or 0.005E02). It has been suggested that verification may be performed by restricting representation to a common form.

### 3.5.7 Floating Point Types

No further implications for formal verification technology.

### 3.5.8 Operations of Floating Point Types

No further implications for formal verification technology.

### 3.5.9 Fixed Point Types

Fixed point types are decreased in accuracy as a result of operations. The problems of undefined inaccuracy pertinent to real types are not applicable in fixed point types, and measurement of maximum error is possible through standard numerical methods techniques.

### 3.5.10 Operations of Fixed Point Types

No further implications for formal verification technology.

## 3.6 Array Types

It has been suggested [McGettrick] that inclusion of some array-specific functions would be beneficial in verifying programs containing arrays. These functions would include "select" (for identifying a specific array element), "assign" (for updating an array element), and boolean functions to indicate the ordering of an array.

### 3.6.1 Index Constraints and Discrete Ranges

No further implications for formal verification technology.

### 3.6.2 Operations of Array Types

No further implications for formal verification technology.

### 3.6.3 The Type String

No further implications for formal verification technology.

## 3.7 Record Types

All restrictions and implications inherent in the types of a component of a record are implicit for that component.

### 3.7.1 Discriminants

No further implications for formal verification technology.

### 3.7.2 Discriminant Constraints

No further implications for formal verification technology.

### 3.7.3 Variant Parts

No further implications for formal verification technology.

### 3.7.4 Operations of Record Types

No further implications for formal verification technology.

## 3.8 Access Types

The access type in Ada, as mentioned before, is roughly equivalent to the pointer type in Pascal [Young81]. Verification systems, such as the Stanford Pascal Verifier [Luckham79] have been developed for dealing with such types using formal verification technology, and [McGettrick] provides a notation and an axiomatization specifically for the access types as found in Ada. The major area of difficulty with access types and corresponding objects are in the areas where aliasing might occur [Odyssey85, Tripathi] in parameter passing. This issue is discussed below in

Operations of Access Types, Section 3.8.2 and Subprogram Calls, Section 6.4.

Some researchers [Odyssey85] have recommended forbidding access types to task types. The reason for this is twofold: 1. To prevent the dynamic creation of tasks. 2. The lack of research information passing tasks as parameters in subprogram calls.

### 3.8.1 Incomplete Type Declarations

No further implications for formal verification technology.

### 3.8.2 Operations of Access Types

The major concern in the use of access types is the possibility of aliasing (see [Odyssey85] for a lengthy discussion of the matter). One possible solution to the aliasing problem with access types, presented in [Tripathi], is to define a new operator for access types which performs component copying, rather than pointer duplication. This solution is appealing with the advent of the evaluation of the Ada language, due in the latter part of the 1980s, when changes and updates based on several years of working experience with the language will be incorporated into the language. However, restrictions on parameter passing [Odyssey85, Young81] would appear to provide the same benefit with fewer changes.

### 3.9 Declarative Parts

No further implications for formal verification technology.

## 4. Names and Expressions

This chapter of the LRM deals with the use of identifiers as names, combining names into expressions, and rules for evaluation of both names and expressions. The areas of interest from the verification viewpoint include the manner of expression evaluation (Section 4.5), accuracy of operations on real types (Section 4.5.7), and allocators (Section 4.8).

### 4.1 Names

No further implications for formal verification technology.

### 4.1.1 Index Components

No further implications for formal verification technology.

### 4.1.2 Slices

No further implications for formal verification technology.

### 4.1.3 Selected Components

No further implications for formal verification technology.

### 4.1.4 Attributes

No further implications for formal verification technology.

## 4.2 Literals

No further implications for formal verification technology.

## 4.3 Aggregates

No further implications for formal verification technology.

### 4.3.1 Record Aggregates

No further implications for formal verification technology.

### 4.3.2 Array Aggregates

No further implications for formal verification technology.

## 4.4 Expressions

No further implications for formal verification technology.

## 4.5 Operators and Expression Evaluation

Expressions, particularly numeric expressions, raise the possibility of exceptions where either intermediate or final results may be outside the bounds of the type of the object which is the target of the expression. If the expression is successfully evaluated, and the result is outside the bounds of the target of the assignment, then the exception constraint_error will be raised. Handling this exception is within the scope of today's verification technology. However, the effect on verification of an intermediate result which is outside the bounds of the arithmetic type is much greater. An intermediate calculation may either raise a numeric_error constraint or result in numeric overflow. While an expression may be completely computable (in terms of universal arithmetic), it may not be actually reliable on a particular implementation. As an example of this consider the expression:

```
IntNo:= (MAXINT*MAXINT)/MAXINT;
```

This is equivalent, mathematically, to:

```
IntNo:= MAXINT;
```

The result should be MAXINT. However an intermediate result, MAXINT*MAXINT, is too large to be contained in the machine and may result in an overflow condition. Verification concerning such expressions would thus be dependent upon knowing the implementation limitations, and proofs would have to take this fact into consideration.

### 4.5.1 Logical Operators and Short-Circuit Control Forms

No further implications for formal verification technology.

### 4.5.2 Relational Operators and Membership Tests

No further implications for formal verification technology.

### 4.5.3 Binary Adding Operators

No further implications for formal verification technology.

### 4.5.4 Unary Adding Operators

No further implications for formal verification technology.

### 4.5.5 Multiplying Operators

No further implications for formal verification technology

### 4.5.6 Highest Precedence Operators

No further implications for formal verification technology.

### 4.5.7 Accuracy of Operations with Real Operands

As mentioned previously (Sections 3.5.6 - 3.5.10), verification of the accuracy of operations on real numbers is currently beyond the state of the art in formal verification technology. [As a demonstration of this, consider

```
J := (1.0/3.0) + (1.0/3.0) + (1.0/3.0)
```

Since each addend is equal to .33333.... (as represented on the machine being used) the result would be .99999..... The result however should be simply 1. Any later operations using the variable J would contain a degree of error and would result in multiplication of the error. Since the accuracy is dependent upon the implementation, its verification is currently beyond the state of the art.]

## 4.6 Type Conversions

No further implications for formal verification technology.

## 4.7 Qualified Expressions

No further implications for formal verification technology.

## 4.8 Allocators

In the use of allocators, care must be taken to see that the initialization recommendations (in Section 3.2.1) are taken into consideration in order to prevent programs from operating on objects that are not initialized.

## 4.9 Static Expressions and Static Subtypes

No further implications for formal verification technology.

## 4.10 Universal Expressions

No further implications for formal verification technology.

## 5. Statements

This chapter of the LRM describes the eight kinds of statements in Ada. These are assignment (with special case for arrays), conditional (if), case, loop, block, exit, return, and goto. For the most part, these are the standard kinds of statements found in most modern day programming languages. As a result, when taken in isolation, these constructs are amenable to known methods in formal verification technology [McGettrick].

This chapter is being considered only from the sequential perspective; the effects of parallel execution are considered in Chapter 9, "Tasks". Proof rules for assignments to scalar elements are a straightforward substitution of the assigned value into the precondition to generate the postcondition. This becomes more complex when updating a single array element, since most of the array values are unchanged. The compilation is adequately severe that [Mills86] proposes elimination of arrays as data structures. For **if** statements, there is a single precondition but there are two postconditions, one for each the "then" and "not then" (possible "else") branch. After an **if** statement, it is necessary to show that both postconditions logically imply the next precondition. The **case** statement is an extension of the if statement. **Loop** statements require the development of a loop invariant. An invariant is an assertion that is typically the strongest statement that is true at particular point of the loop; it frequently contains all of the variables of the loop. The statements within the loop are verified in the normal fashion, except that the loop invariant is logically "ANDed" to the precondition and postcondition at the point of interest of the invariant. Also, at the conclusion of the loop, the invariant is true and the controlling Boolean condition is false. The **block** statement is discussed under visibility; the **exit** and **return** statements are covered in Chapter 6, "Subprograms".

## 5.1 Simple and Compound Statements - Sequences of Statements

Rules for the verification of simple and compound statements are standard fare in formal verification technology. The proof rules given in [McGettrick] are sufficient for most types of statements if the following criteria are met: (1) no aliasing, (2) no side effects, and (3) no nonlocal variables. These criteria are good software engineering practice, and as such should not require extra programming effort. These criteria are necessary in that each of them, if not met, may cause an unintended change to be effected during the course of the execution of the software. These unintended changes may cause an erroneous, or at least unexpected, result that would preclude the verification of this software. For the most part, the set of statement constructs available in Ada is similar to those found in Algol, Pascal and Gypsy, languages for which axiomatizations and proof rules of these constructs are fairly well known and understood.

## 5.2 Assignment Statement

No further implications for formal verification technology.

### 5.2.1 Array Assignments

No further implications for formal verification technology.

## 5.3 If Statements

No further implications for formal verification technology.

## 5.4 Case Statements

No further implications for formal verification technology.

## 5.5 Loop Statements

No further implications for formal verification technology.

## 5.6 Block Statements

No further implications for formal verification technology.

## 5.7 Exit Statements

No further implications for formal verification technology.

## 5.8 Return Statements

No further implications for formal verification technology

## 5.9 Goto Statements

No further implications for formal verification technology. Although it would be considered bad form (and has been recommended against in [Odyssey85]) to make widespread use of the goto statement, particularly due to the additional complexity it would cause in verification, it has been shown [McGettrick] that verification of such constructs is not intractable.

# 6. Subprograms

This chapter of the LRM defines the mechanism for describing subprograms (technically procedures and functions), the mechanisms for their invocation, and the manner of parameter passing. In the absence of concurrence, as with other aspects of the Ada language, most aspects of subprogram declaration and invocation are equivalent with respect to verification as those in other languages. Some specific differences that are relevant in the application of formal verification technology to Ada are the issues of subprogram declarations (6.1), formal parameter modes (6.2), and parameter "aliasing" (6.4).

The major issues in verification of subprograms are all related to parameters and parameter passing. Verifying the subprogram requires establishing a precondition that, if true prior to execution of the subprogram, assures that a postcondition is true upon completion of the subprogram's execution. Verifying a specific subprogram call requires verifying certain conditions about the parameters involved in the call. These parameters fall into one of two categories, input parameters or output parameters. Input parameters are used only for passing values to the subprogram; output parameters may have their values altered by the subprogram. The conditions that must be verified for each call are as follows. No variable, either input or output, may appear in either the precondition or postcondition. No formal parameter that appears in the output parameter list may appear more than once in that list, and, no output parameter may appear as an input parameter. The formal condition results in updating multiple variables when only one is intended to be updated. For example, if two subprogram formal parameters, A and B, are both passed variable X through a subprogram call, the result of the statements

```
A:=0
B:=1
```

leaves variable X with the value 1 and no variable from the call with value 0. The postcondition after these two statements would assume the existence of two distinct parameters, one with value 1 the other with value 0. If an output parameter appears as an input parameter, the time at which the input parameter is evaluated becomes critical. If the output variable is updated prior to the evaluation of the input parameter, the value of the input parameter may differ from the value recorded if the output parameter is not updated prior to evaluation of the input parameter. A single actual parameter used for more than one formal parameter is known as "aliasing." Aliasing is discussed in Section 6.4.1.

## 6.1 Subprogram Declarations

Although [Odyssey85] recommends restricting a subprogram from containing another equivalently named subprogram with the same parameter type profile, this restriction is not necessary from the viewpoint of verification, as the scoping rules in Ada clearly define which subprogram is visible at any point in the program text. It might be justified from a human viewpoint, however, as the additional complexity of overloaded names could cause difficulty both in the software development and verification processes.

## 6.2 Formal Parameter Modes

One result of a more formal semantic definition of the language would include the definition of the default values for out parameters, akin to that described in Section 4.2.1 for object declarations [Odyssey85]. Doing so would help prevent erroneous programs.

Another difficulty, due to the lack of specificity on the part of the language designers, is the method of parameter passing, and the manner in which this affects verification. It is quite elementary, as shown in [Tripathi], to concoct an example in which a legal program can generate different results based upon whether the parameter passing mechanism chosen is call by value or call by reference. Again, the completion of the formal semantic definition would address such an issue. Also, regardless of the parameter passing mode, verifying programs with aliased subprogram calls is intractable.

### 6.3 Subprogram Bodies

No further implications for formal verification technology.

#### 6.3.1 Conformance Rules

No further implications for formal verification technology.

#### 6.3.2 Inline Expansion of Subprograms

No further implications for formal verification technology.

### 6.4 Subprogram Calls

#### 6.4.1 Parameter Associations

The association of parameters at subprogram call points would be the ideal location to exclude aliasing [Good80, Odyssey85]. Although there might be a loss of efficiency, the fact that aliasing is unnecessary and complicates application of formal verification technology [Young81] would seem to be sufficient reason for its elimination.

Also, the matter of indeterminacy, inherent in the language definition, such as 6.4(6) (quoted below) must be resolved in the formal semantic definition in order to allow both the presentation of proof techniques and the development of automated tools for supporting proofs.

6.4 (6) Subprogram Calls

> The parameter associations of a subprogram call are evaluated in some order that is not defined by the language. Similarly, the language rules do not define in which order the values of in out or out are copied back into the corresponding actual parameters (when this is done).

#### 6.4.2 Default Parameters

No further implications for formal verification technology.

## 6.5 Function Subprograms

If function subprograms are truly functional, they will not include any input or output, nor will they reference nonlocal (i.e., global) variables. If this is the case, Ada function subprograms can be verified similarly to Gypsy function subprograms, in which these restrictions are enforced by the language.

## 6.6 Parameter and Result Type Profile - Overloading of Subprograms

No further implications for formal verification technology.

## 6.7 Overloading of Operators

No further implications for formal verification technology.

## 7. Packages

This chapter of the LRM deals with the specification of packages as a means to encapsulate data and subprograms into a single structure. Of particular interest in the application of formal verification technology are the use of private and limited types within a package.

The use of packages directly supports the notion of abstract data types [Gerhart, Guttag, Shaw], a common abstraction mechanism in software engineering used to reduce program complexity. Proof rules for dealing with packages, including package invariants (similar to loop invariants) have been outlined [Cohen]. The package invariant is a formula that is asserted to be true and after each call of the package's subprograms. The application of formal verification techniques at the level of packages supports the privacy principle [Tripathi], and might be sufficient to allow implementations of kernel based security monitors [TCSEC].

## 7.1 Package Structure

No further implications for formal verification technology.

## 7.2 Package Specifications and Declarations

No further implications for formal verification technology.

## 7.3 Package Bodies

No further implications for formal verification technology.

## 7.4 Private Type and Deferred Constant Declarations

No further implications for formal verification technology.

### 7.4.1 Private Types

The declaration of a type as private in Ada is similar to the declaration of a type as an abstract type in Gypsy. Use of these types are possible only through specific operations on the type, while the underlying implementation (called the private part in Ada, and the concrete type in Gypsy) is not available to the programmer using the type. This provides a wonderful opportunity for modularization both of software development and of program proofs. Since the user of these types and operations is unable to know the underlying representation, it is not possible to develop programs or proofs that depend upon that representation, thus making them independent of it.

### 7.4.2 Operations of a Private Type

No further implications for formal verification technology.

### 7.4.3 Deferred Constants

### 7.4.4 Limited Types

A limited type provides the same benefits as a private type with respect to formal verification technology, as well as aiding further in the support of package and type invariants [Cohen]. Since assignment and tests for equality, are further restricted, an implementation in which two underlying objects might have different concrete representations but be equivalent from the abstract point of view (just as one half and two quarters are equivalent) could be supported. This feature provides additional support for a wider range of abstract data type implementations.

## 7.5 Example of a Table Management Package

No further implications for formal verification technology.

## 7.6 Example of a Text Handling Package

No further implications for formal verification technology

## 8. Visibility Rules

This chapter of the LRM deals with the rules for determining the visibility of names and identifiers in the Ada program text. For the most part, such rules are applicable at the syntactic and semantic phases of the analysis of the Ada program text, although they will be relevant during verification to determine the scope of variables being used in proofs of theorems. One area in which the visibility rules of Ada do have an impact upon the application of formal verification technology is the allowance of nested subprogram declarations. Although allowed in the language, [Young81] points out that use of such constructs is easily replaced through the use of good modular design and proper parameter structuring. This is not to say that a program containing nested subprograms cannot be verified, and one that does not contain nested subprograms can; however, the application of verification technology to the programs that are developed in a more modularized manner is a less complicated process, since modular programs aid in the development of modular proofs.

### 8.1 Declarative Region

No further implications for formal verification technology.

### 8.2 Scope of Declarations

No further implications for formal verification technology.

### 8.3 Visibility

No further implications for formal verification technology.

### 8.4 Use Clauses

No further implications for formal verification technology.

### 8.5 Renaming Declarations

No further implications for formal verification technology.

### 8.6 The Package Standard

No further implications for formal verification technology.

### 8.7 The Context of Overload Resolution

No further implications for formal verification technology.

## 9. Tasks

This chapter of the LRM deals with the concurrent aspects of the Ada language. Far and away, concurrence is one of the most hazardous obstacles in the way of applying formal verification technology to the Ada language. By allowing only restricted use of tasking, it appears that concurrence in Ada can be made amenable to application of formal verification technology; however, it remains to be seen whether what remains has any semblance to what would be called Ada, and whether it would have any usefulness for the objectives for which it was designed into the language. Use of the techniques employed in Communicating Sequential Processors (CSP) [Barringer] appears to be a promising possibility while other research indicates that restriction of communication to only buffers (a la Gypsy) [Young80], to only scalars [Odyssey85], or to only those entry points in which pre-condition and postcondition assertions have been specified [Tripathi], would alleviate many of the inherent difficulties in applying formal verification technology to the Ada concurrence problem.

It must be noted that these restrictions to communicating between tasks reflect the state of the art rather than assess feasibility. Although no one has published proof rules for passing aggregate types, for example arrays or records, development of such proof rules seems quite feasible.

Several researchers [Odyssey85, Pneuli] have recommended that access pointers to tasks not be allowed. The intent of this restriction is to disallow dynamic creation of tasks. They have also indicated that in the absence of dynamic creation of tasks, proof rules can be obtained for tasking. However, it is unlikely that this limitation will be readily accepted, particularly in the systems programming arena.

The approach used by [Owicki and Gries] to verify Communicating Sequential Processes, Hoare's language framework for concurrent programming, is readily adapted to verification of Ada tasks. This approach consists of two distinct steps: internal verification and external verification. Internal verification consists of proving the task an isolated, sequential program. External verification consists of proving that, with the exception of entries, tasks do not affect any subprograms, tasks or variables declared outside of the task being verified. External verification also requires proof that the task in question is not affected by any subprograms, tasks or variables declared outside of the task; again, entries are the exception to this rule. External verification is performed in two stages:

a) I-O assertions on entries are made, and shared variables are restricted, and

b) A proof against deadlocks and starvation is made.

Deadlock and starvation avoidance proofs are prevalent throughout parallel processing literature.

The verification of tasks also assumes the following:

a) All constructs terminate normally;

b) Subprogram calls have no side effects;

c) Assignments have no side effects, and

d) Tasks may not be aliased.

## 9.1 Task Specifications and Task Bodies

This section of the LRM delineates the mechanism for separating task specifications and bodies. As such, it creates no further implications for formal verification technology.

## 9.2 Task Types and Task Objects

No further implications for formal verification technology.

## 9.3 Task Execution - Task Activation

No further implications for formal verification technology.

## 9.4 Task Dependence - Termination of Tasks

No further implications for formal verification technology.

## 9.5 Entries, Entry Calls, and Accept Statements

Assertions must be made on all I-O entry calls.

## 9.6 Delay Statements, Duration, and Time

There has been very little success in the application of formal verification technology to the area of real-time features. This is due in part to the lack of a formal basis for discussion of time, as well as the inability to deal with factors such as operating system overhead, change in speed of central processors, and code optimizations. Use of delays in order to achieve certain synchronization between actions would much more preferably be accomplished through semaphores and inter-task communications.

## 9.7 Select Statements

### 9.7.1 Selective Waits

The selective wait is another area in the language that provides for indeterminacy:

### 9.7.1(5) Selective Waits

... are evaluated in some order that is not defined by the language; ...

One result of this specification is that selective waits that contain an else clause can be correctly implemented by a compiler that always chooses the else clause. It will thus be necessary to restrict the use of selective waits with else clauses in order to assure avoidance of such anomalous behavior.

### 9.7.2 Conditional Entry Calls

Conditional entry calls are similar to selective waits, in that they can specify an else clause to be performed. However, the execution of the else part of the conditional entry call only occurs after the conditional entry has failed, which does not lead to indeterminacy, and can thus be handled without difficulty.

### 9.7.3 Timed Entry Calls

As mentioned above, dealing with real-time aspects of programs is beyond the state of the art of formal verification technology. Thus, the use of time entry calls would have to be forbidden.

### 9.8 Priorities

The specification of task priority is a mechanism that can be used to indicate relative importance of concurrent tasks with respect to scheduling of processing. However, the LRM clearly indicates that this area provides an indeterminacy. For tasks of the same priority, the scheduling order is not defined by the language, and the LRM explicitly discourages the use of priority assignment for synchronization. Since priorities should be used only to indicate relative degrees of urgency and not for synchronization, synchronization should be used rather than priorities.

### 9.9 Task and Entry Attributes

Tasks and entries have three attributes as specified in the LRM: T'CALLABLE, T'TERMINATED, and E'COUNT. It has been recommended [Odyssey85] that the use of these attributes be restricted. The reason behind this recommendation is that use of these dynamic attributes enables the passing of information in a manner which is much more difficult to keep track of than the normal manner of parameter passing. Since timing and scheduling have already been excluded from the realm under which formal verification technology is applicable, this restriction seems reasonable.

### 9.10 Abort Statements

No further implications for formal verification technology.

### 9.11 Shared Variables

Shared variables are the major construct in tasking that will have to be restricted (although perhaps simulated through use of other constructs using synchronization) in order to apply formal verification technology to Ada. On this matter, there is no disagreement among the researchers[Cohen, Good80, Odyssey85, Tripathi].

### 9.12 Example of Tasking

No further implications for formal verification technology.

## 10. Program Structure and Compilation Issues

This chapter of the LRM describes the units of compilation, attends to the ordering requirements for program libraries, and touches briefly on the results of optimizations. Aside from the results of optimizations (see also Sections 1.6 and 11.7 with respect to exceptions) this part of the LRM is neutral with respect to application of formal verification technology.

## 10.1 Compilation Units - Library Units

No further implications for formal verification technology.

### 10.1.1 Context Clauses - With Clauses

No further implications for formal verification technology.

### 10.1.2 Examples of Compilation Units

No further implications for formal verification technology.

## 10.2 Subunits of Compilation Units

### 10.2.1 Examples of Subunits

No further implications for formal verification technology.

## 10.3 Order of Compilation

No further implications for formal verification technology. It is known that proof order is dependent upon modifications in the same manner as compilation order, and this provides no new difficulties.

## 10.4 The Program Library

No further implications for formal verification technology.

## 10.5 Elaboration of Library Units

No further implications for formal verification technology.

## 10.6 Program Optimization

The LRM allows optimizations to be performed in situations where execution of the code would be known to raise exceptions at runtime (such as an expression causing division by zero). This causes no impedance to the application of verification technology so long as the semantics embodied in the proof techniques and tools recognize the same situation, and proceed in the same manner with respect to symbolic evaluation and/or path analysis in generating the relevant verification conditions.

# 11. Exceptions

This chapter of the LRM defines the exception constructs and mechanisms and rules of handling exceptions within programs. As mentioned earlier in Section 1.6, exceptions have been discussed widely in the general literature [Goodenough, Liskov] as well as with specific respect to the Ada language [Luckham80, Young81]. Other authors have discussed the relation of exceptions with respect to other aspects of the language [Cohen, Odyssey85, Tripathi].

The major difficulty with exceptions [Tripathi] in the Ada language from the point of view of verification is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during symbolic execution of programs during verification. This complexity is furthered by the fact that exceptions are propagated "as is," which could cause an unhandled exception to propagate from several levels down to a routine that has no understanding of the meaning of the exception. A stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions - conversion of unhandled exceptions to some ROUTINE_ERROR on exit from a block (within a package or not), or explicit use of "OTHERS" clauses at all possible functions (not a convenient approach), the complexity could be reduced.

Another matter of concern with respect to exceptions is due to the non-specificity of the language with respect to modes of parameter passing. If a compiler passes an IN OUT parameter by copy on entry and on exit, the actual parameter may never be updated if the routine raises an exception, whereas if the parameter is passed by reference, changes to the actual parameter may actually change the passed formal parameter, and the value will have been updated in the presence of a raised exception.

## 11.1 Exception Declarations

No further implications for formal verification technology.

## 11.2 Exception Handlers

No further implications for formal verification technology.

## 11.3 Raise Statements

No further implications for formal verification technology.

## 11.4 Exception Handling

Exceptions in Ada are handled by the innermost execution frame or accept statement enclosing the statement that caused the exception. (Exceptions within accept statements are discussed in Section 11.5.) Although the Ada mechanism for propagating exceptions is dynamic, there is no clear evidence that they are intractable from a verification standpoint [Young81].

### 11.4.1 Exceptions Raised During the Execution of Statements

The major difficulty with exceptions [Tripathi] in the Ada language from the point of view of verification is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during symbolic execution of programs during verification. This complexity is further by the fact that exceptions are propagated "as is," which could cause an unhandled exception to be propagate from several levels down to a routine that has no understanding of the meaning of the exception. A stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions - conversion of unhandled exceptions to some ROUTINE_ERROR on exit from a block (within a package or not), or explicit use of "OTHERS" clauses at all possible junctions (not a convenient approach) - the complexity could be reduced.

Another matter of concern with respect to exceptions is due to the non-specificity of the language with respect to modes of parameter passing. If a compiler passes an IN OUT variable by copying on entry and on exit, the variable may never be updated if the routine raises an exception, whereas if the variable is passed by reference, changes to the local variable may actually change the passed variable, and the value will have been updated in the presence of a raised exception. (See example are below.)

```
with text_io; use text_io;
procedure arr is

     package int_io is new text_io.integer_io (integer); use
          int_io;
     type x is array (1.. 10) of integer);

     y : x := (others = >0);

     procedure bar (a : in out x) is
          i : integer := 1;
          begin
               loop
                  a (1) := a(1) + 1;
                  i := i * 100;
               end loop;
          end bar;
      begin
          bar (y);
          exception
          when others = put (y (1));
                        new_line;
     end arr;
```

## 11.4.2 Exceptions Raised During the Elaboration of Declarations

No further implications for formal verification technology.

## 11.5 Exceptions Raised During Task Communication

Exceptions raised during task communication are complicated more by the difficulty in dealing with tasking in Ada than in dealing with exceptions. However, it is likely that a specification language can be developed to adequately describe the behavior of exceptions within tasking to accommodate formal verification technology.

## 11.6 Exceptions and Optimization

As mentioned in Section 10.6, optimization is really a compiler issue, and not a verification issue. Provided that the compiler adheres to the semantics of the language, it is free to perform whatever optimizations may be desired to increase the runtime efficiency of any program that raises the exception in a much shorter time frame, without any adverse implications for verification.

## 11.7 Suppressing Checks

The specification of suppression of runtime checks provides no difficulty if such directives are also provided as information during the verification process. This is necessary to prevent programs from becoming erroneous (see LRM 1.7 (20)). However, a properly integrated verification and compilation environment can determine at what points in a program the runtime checks can be suppressed, and perform the same benefit [McHugh]. The application of verification technology should preclude the possibility of erroneous programs.

Suppression of runtime checks through the use of the suppress pragma directive to the compiler is essentially a change in the semantics of the language. For software in which it is desired to use this feature, it would be necessary to re-verify the software with different symbolic evaluation and verification condition generation.

One very useful side benefit of applying formal verification technology to software development is the possibility of proving that certain checks need not be made, and knowing that the suppress pragma can indeed be utilized without re-verification [McHugh]. This would be applicable only over small segments of program code, but could have significant payback in execution speed.

## 12. Generic Units

This chapter of the LRM describes the structure and application of generic units within Ada. The use of generic constructs is one of the more novel innovations in the Ada language, and as such, little effort in the formal verification technology realm had previously been applied to this area.

Several efforts [McGettrick, Tripathi, Young80] have investigated the applicability of formal verification technology to this construct, and found it amenable to such methods. One major question that remains to be resolved is whether generics can be proven prior to instantiation, or whether it is necessary to reprove them for each instantiation. Although the former is preferable, the latter may be necessary given some generics and their functional subprograms [Tripathi]. The major difficulty lies in the validity of precondition and postcondition assumptions used during proofs of generics when given particular instantiations.

A simple solution to providing generics is to prove each instantiation of the generic (e.g., each instantiation of a generic subprogram would be proven as a subprogram). This solution is costly and defeats the intended purpose of generics; a one time verification scheme is a preferred alternative. This "one shot" scheme would utilize a method similar to a standard proof routine with several key restrictions. These restrictions are as follows:

The standard routine must be capable of verifying constructs on a modular basis

Proof rules utilized during the proof must be generic within the scope of the generic parameter declaration

A resultant specification of the generic must be performed so that its effects on external constructs may be used for the verification of the external constructs.

## 12.1 Generic Declarations

No further implications for formal verification technology.

### 12.1.1 Generic Format Objects

No further implications for formal verification technology.

### 12.1.2 Generic Formal Types

No further implications for formal verification technology.

### 12.1.13 Generic Formal Subprograms

No further implications for formal verification technology.

## 12.2 Generic Bodies

No further implications for formal verification technology.

## 12.3 Generic Instantiation

No further implications for formal verification technology.

### 12.3.1 Matching Rules for Formal Objects

No further implications for formal verification technology.

### 12.3.2 Matching Rules for Formal Private Types

No further implications for formal verification technology.

### 12.3.3 Matching Rules for Formal Scalar Types

No further implications for formal verification technology.

### 12.3.4 Matching Rules for Formal Array Types

No further implications for formal verification technology.

### 12.3.5 Matching Rules for Formal Access Types

No further implications for formal verification technology.

### 12.3.6 Matching Rules for Formal Subprograms

## 12.4 Example of a Generic Package

No further implications for formal verification technology.

# 13. Representation Clauses and Implementation-Dependent Features

This chapter of the LRM deals with implementation-specific matters at such a low level as to almost preclude application of formal verification technology. Several of the constructs, such as representation clauses, length clauses, enumeration representation clauses, and address clauses are on the order of specific directives to the compiler and would have no noticeable effect on the execution of the resulting program. Machine code insertions, and interfaces to subprograms written in other languages man be amenable to application of formal verification technology given an adequate specification language, while unchecked programming constructs are clearly beyond the scope of the current state of the art, and would have to be disallowed.

## 13.1 Representation Clauses

No further implications for formal verification technology.

## 13.2 Length Clauses

No further implications for formal verification technology.

## 13.3 Enumeration Representation Clauses

No further implications for formal verification technology.

## 13.4 Record Representation Clauses

No formal implications for formal verification technology.

## 13.5 Address Clauses

The use of the address clauses must be restricted so as to preclude the use of a particular address (whether it be a device, or actual memory) as a shared variable. An example of a program fragment which allows this follows:

```
with system; use system;

procedure adr is

     a, b : system.address:
     for a use at no_addr:      -- NO_ADDR : constant ADDRESS:
     for b use at no_addr:      -- from package SYSTEM

begin
     null;
end addr;
```

If a and b are each passed to different tasks, it then becomes possible to cause an implicit sharing of variables. (Given some vendor's SYSTEM package, it might by possible to generate any desired address for this example.)

### 13.5.1 Interrupts

The example in 13.5 is as applicable for interrupts as it is for shared memory.

## 13.6 Change of Representation

No further implications for formal verification technology.

## 13.7 The Package System

No further implications for formal verification technology.

### 13.7.1 System-Dependent Named Numbers

### 13.7.2 Representation Attributes

No further implications for formal verification technology.

### 13.7.3 Representation Attributes of Real Types

No further implications for formal verification technology.

## 13.8 Machine Code Insertions

The difficulty with allowing machine code insertions from a verification point of view is the inability to correlate the specification of the machine code instructions with the intended abstract behavior at the Ada language level. If it is possible to specify the intended behavior, it would likely be preferable (from the verification viewpoint) to program in Ada; if not, attempting to use such insertions would stymie the verification process.

One possible use of this feature would be to insert calls to currently existing functions (e.g., sort routines), that might be known to already work, rather than having to recode the routines in Ada. A specification of the routines at the Ada specification language level and "trusting" the routines to not stray from their specified behavior might provide an acceptable temporary compromise.

## 13.9 Interface to Other Languages

Interfaces to other languages are like interfacing to machine language insertions - the major difficulty lies in the specification of the intended behavior of the interfaced routines, and in the proof that the routines do indeed perform the specified behavior. Calling a routine that performs a cube root of its argument a square root program and specifying it as such will simply not cause it to perform a square root function, and can lead to "proven" incorrect programs if such routines are trusted to such an extent.

## 13.10 Unchecked Programming

The use of unchecked programming should be generally disallowed in order to allow the application of formal verification technology.

### 13.10.1 Unchecked Storage Deallocation

Since the only program-visible effect of using unchecked_deallocation is the assignment of the access value null to the variable being deallocated, there is no problem with the use of this feature from the correctness of the most Ada programs-simply treat such calls as assignments to null. If, however, it is necessary (and possible) to specify something concerning the amount of available storage before and after such calls, then something must be known about the underlying representation, and the specification language must have some mechanism for specifying such attributes as available storage space.

### 13.10.2 Unchecked Type Conversions

One major difficulty with the use of unchecked type conversions is specifying the transformation between the two types that takes place during the conversion. As an example (see example convert below), conversion from an eight character string to a sixty-four element boolean array with the Verdix compiler causes the boolean array to contain representations of the corresponding ASCII characters with the least significant bit first. Other compiler implementations may store the data with the most significant bit first, and generate different results.

```
with text_io; use text_io;
with unchecked conversion;
procedure convert is

     subtype string8 is string (1 .. 8);
     type bit64 is array (1 .. 64) of boolean;
     pragma pack (bit64);

     s : string8 := "12345678";
     b : bit64;

     function string8tobit64 is new unchecked_conversion
     (string8, bit64);

     begin
          b := string8tobit64 (s);
          for i in 0 .. 7 loop
               for j in 1 .. 8 loop

               if b (i * 8 + j) then
                  put ("1");
               else
                  put ("0");
               end if
          end loop;
          new_line;
     end loop;
     end convert;
```

14. Input-Output

This chapter of the LRM describes the mechanisms for input and output from an Ada program and the management of file objects. The packages described include procedures for the input of sequential, direct, and numeric data. If implemented correctly, and as intended by the original programmer, these procedures will not preclude verification. However, if implemented incorrectly, the results obtained are unpredictable.
This is a problem of semantics, and not a problem of verification.

The major obstacles to application of formal verification technology that input and output creates are the lack of the semantics of input and output, and the ability to do input and output anywhere within an Ada program. Restricting functions from having side effects, such as input and output, is encouraged [Odyssey85, Tripathi], and the use of specifications for those procedures which do have input and output is a necessity for verification.

### 14.1 External Files and File Objects

No further implications for formal verification technology.

### 14.2 Sequential and Direct Files

No further implications for formal verification technology.

#### 14.2.1 File Management

No further implications for formal verification technology.

#### 14.2.2 Sequential Input-Output

No further implications for formal verification technology.

#### 14.2.3 Specification of the Package Sequential_IO

No further implications for formal verification technology.

#### 14.2.4 Direct Input-Output

No further implications for formal verification technology.

#### 14.2.5 Specification of the Package Direct_IO

No further implications for formal verification technology.

### 14.3 Text Input-Output

No further implications for formal verification technology.

#### 14.3.1 File Management

No further implications for formal verification technology

#### 14.3.2 Default Input and Output Files

No further implications for formal verification technology.

#### 14.3.3 Specification of Line and Page Lengths

No further implications for formal verification technology.

#### 14.3.4 Operations on Columns, Lines, and Pages

No further implications for formal verification technology.

#### 14.3.5 Get and Put Procedures

No further implications for formal verification technology.

14.3.6 Input-Output of Characters and Strings

No further implications for formal verification technology.

14.3.7 Input-Output for Integer Types

No further implications for formula verification technology.

14.3.8 Input-Output for Real Types

No further implications for formal verification technology.

14.3.9 Input-Output for Enumeration Types

No further implications for formal verification technology.

14.3.10 Specification of the Package Text_IO

No further implications for formal verification technology.

14.4 Exceptions in Input-Output

No further implications for formal verification technology.

14.5 Specification of the Package IO_Exceptions

No further implications for formal verification technology.

14.6 Low Level Input-Output

No further implications for formal verification technology.

14.7 Example of Input-Output

No further implications for formal verification technology.

BIBLIOGRAPHY:

Ada Programming Language, ANSI-MIL-STD-1815A, Department of Defense, 22 January 1983.

Barringer, H., Mearns, I., "Axioms and Proof Rules for Ada Tasks", IEEE Proceedings, Volume 29(E), Number 2, pp. 38-48, March 1982.

Berg, H.K., Boebert, W.E., Franta, W.R., Moher, T.G., Formal Methods of Program Verification and Specification, Prentice-Hall, Inc., Englewood Cliffs, NY 07632, 1982.

Cohen, Norman H., "Ada Axiomatic Semantics: Problems and Solutions", SofTech, Inc., One Sentry Parkway, Suite 6000, Blue Bell, PA 19422-2310, May 1986.

Dahl, Ole-Johan, Can Program Proving Be Made Practical?, Institute of Informatics, University of Oslo, Norway, 1978.

DiVito, Ben, "A Verifiable Subset of Ada", TRW, unpublished manuscript.

The Draft Formal Defintion of ANSI-MIL-STD 1815A Ada, EEC Multiannual Programme, Project No. 782, Annex 1, Version 14-12-1984, Dansk Datamatik Center, Lundtoftevej 1C, DK-2800 Lyngby, Denmark.

Ernst, G.W., and Hookway, R.J., "Specification and Verification of Generic Program Units in Ada", Department of Computer Engineering and Science, Case Institute of Technology, Case Western University, Cleveland, Ohio.

Floyd, R.W., "Assigning Meanings to Programs", Proceedings of the American Mathematical Society Symposia in Applied Mathematics, Vol. 19, pp. 19-31.

Gerhart, Susan L., "Fundamental Concepts of Program Verification", AFFIRM Memo-15-SLG, University of Southern California Information Science Institute, Marine Del Rey, CA 90291, February 18, 1980.

Gerth, R. "A Sound and Comple Hoare Axiomatization of the Ada Rendezvous", Proceedings of the 9th International Coloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 140, Springer Verlag, pp. 252-264, 1982.

Gerth, R. and deRoever, W.P., "A Proof System for Concurrent Ada Programs", RUU-CS-83-2, Rijksuniversiteit Ultrecht, January 1983.

Good, Donald I. et al., "Report on the Language Gypsy, Version 2.0," The University of Texas at Austin, ICSCA-CMP-10, September 1978.

Good, Donald I., Cohen, Richard M., and Keeton-Williams, James, "Principles of Proving Concurrent Programs in Gypsy", Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX 78712, January 1979.

Good, Donald I. et al., "An Evaluation of the Verifiability of Ada", September 1980.

Goodenough, John B., "Exception Handling: Issues and a Proposed Notation", Communications of the ACM, 18(12):683-696, December 1975.

Guttag, John V., Horowitz, Ellis, and Musser, David R., "Abstract Data Types and Software Validation", Communications of the ACM, 21(12), December 1978.

Hantler, Sidney L., and King, James C., "An Introduction to Proving the Correctness of Programs", Computing Surveys, Vol. 8, No. 3, pp. 331-353.

Hill, A.D., "Asphodel - An Ada Compatible Specification and Design Language", (unpublished manuscript), Central Electricity Generating Board, Computing and Information Systems Department, Laud House, 20 Newgate Street, London EC1A 7AX, U.K.

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Communications of the ACM, 12(10):576-581, October 1969.

Hoare, C.A.R., and Wirth, Niklaus, "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica, :2, 19876.

Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., Wichmann, B.A., "Preliminary Ada Reference Manual" and "Rationale for the Design of the Ada Programming Language", ACM SIGPLAN Notices, 14(6), June 1979.

Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada, HQ85-29920-1, Institute for Defense Analyses, 1801 N. Beauregard St., Alexandria, VA 22311, May, 1985.

Kemmerer, Richard A., Verification Assessment Study, Final Report, National Computer Security Center, C3-CR01-86, Ft. George G. Meade, MD 20755-6700, March 27, 1986.

Formal Definition of the Ada Programming Language, Institute National de Recherche en Informatique et en Automatique, November, 1980.

Liskov, Barbara H., and Snyder, Alan, "Exception Handling in CLU", IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, November 1979.

Luckham, David C. and Suzuki, "Norihisa, Verification of Array, Record, and Pointer Operations in Pascal", ACM Transactions on Programming Languages and Systems, pp. 226-244, October 1979.

Luckham, David C. and Polak, Wolfgang, "Ada Exception Handling: An Axiomatic Approach", ACM Transactions on Programming Languages and Systems, 2(2):225-233, April 1980.

Luckham, David C., von Henke, Friendrich W., Krieg-Brueckner, Bernd, Owe, Olaf, "ANNA-A Language for Annotating Ada Programs, Preliminary Reference Manual", Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

McGettrick, Andrew D., Program Verification Using Ada, Cambridge Computer Science Texts - 13, Cambridge University Press, 1983.

McHugh, John, "Towards Efficient Code from Verified Programs", Technical Report ICSCA-40, Institute for Computing Science, University of Texas at Austin, March 1984.

Mills, Harlan, "Human Verification in Ada.", Presented at the Third IDA Workshop on Ada Specification and Verification, Research Triangle Park, NC. May 14-16, 1986.

Nyberg, Karl A., Hook, Audrey A., Kramer, Jack F. "The Status of Verification Technology for the Ada Language", Institute for Defense Analyses Paper #P-1859, IDA, 1801 N. Beuregard St., Alexandria, VA 22311, July 1985.

Odyssey Research Associates, Inc., "A Verifiable Subset of Ada", (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, September 14, 1984.

Odyssey Research Associates, Inc., Toward Ada Verification, Preliminary Report (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, March 25, 1985.

Owicki, S.S., and Gries, D., "Verifying Properties of Parallel Programs: An Axiometic Approach." Communications of the ACM Vol. 19, No. 5, May, 1976.

O'Donnell, Michael J., "A Critique of the Foundations of Hoare Style Programming Logics." Communications of the ACM, Vol. 25, No. 12, December 1982, 927-935.

Pneuli, A., and deRoever, W.P., "Rendesvous with Ada - A Proof Theoretical View", Proceedings of the AdaTEC Conference on Ada, Arlington, VA, pp 129-137, October 1982.

Shaw, Mary, "The Impact of Abstraction Concerns on Modern Programming Languages", Carnegie-Mellon University Computer Science Department, Technical Report CMU-CS-80-116, Pittsburgh, PA, April 1980.

Sutherland, David, "Formal Verification of Mathematical Software", NASA Contract Report 172407, Odyssey Research Associates, 1984.

Tripathi, Anand R., Young, William D., Good, Donald I., "A Preliminary Evaluation of Verifiability in Ada", Procedings of the ACM National Conference, Nashville, TN, October 1980.

Young, William D., Good, Donald I., "Generics and Verification in Ada", Proceedings of the ACM Symposium on the Ada Language, Boston, MA, pp. 123-127, 9-11 December 1980.

Young, William D., Good, Donald I., "Steelman and the Verifiability of (Preliminary) Ada", ACM SIGPLAN Notices, 16(2):113-119, February 1981.

## APPENDIX B

## OVERVIEW OF STANFORD UNIVERSITY'S WORK IN FORMAL METHODS APPLIED TO ADA

This Appendix describes the verification effort by the Program Analysis and Verification Group at Stanford University. The effort is analyzed with an eye to its efficiency in serving the needs of the computer security community (particularly the National Computer Security Center) with respect to fielding systems that are verified and accredited at the A1 level.

The information contained in this report has been obtained from the document "An Environment for Ada Software Development Based on Formal Specification" (Stanford TR #CSL-TR-86-305) and from discussions with the staff on the project at Stanford.

## PROJECT OVERVIEW

The current effort is part of a three year project (completion September 1989) to construct a prototype environment of tools for software and hardware development. These tools are based on specification languages with particular emphasis on distributed computing and implemented in the Ada language for maximum portability to various environments. It is hoped that the results of this effort will provide a better understanding into what features are necessary in a development environment with a number of possible applications: requirements analysis and negotiation, rapid prototyping, formal implementation guides, automatic implementations from specifications and construction of self-testing systems.

The effort draws on a significant amount of already completed research, particularly that done in developing the specification languages ANNA (ANNotated Ada) and TSL (Task Sequencing Language), and the effort already performed in developing tools for syntactically parsing the ANNA text and manipulation of the underlying DIANA representation.

The emphasis appears to be similar to that being taken by other researchers, in attempting to apply specification and verification techniques over the entire spectrum of system development, from the top level system requirements and definition down to the low level hardware implementation.

In addition to the primary emphasis on the development of a prototype environment for software and hardware development, additional emphasis is being placed on developing reusable components that may be shared among the various applications level programs in the project. The use of Ada as the implementation language is intended to aid in the portability of the resulting system, and the emphasis on integrability (the commonality of tool interfaces underlying structures, etc.) will also allow the investigation of the utility of such an approach in a large development project. The choice of Ada is also intended to allow the investigation of features of parallelism in specification, runtime checking and software development that have heretofore been unavailable in other environments and with other languages.

## LANGUAGES

This section describes the specification languages (called wide-spectrum in their literature) in use in the Stanford verification project.

### ANNA

ANNA is an annotation language for all constructs of Ada except tasking. Although the language is designed to support various theories of formally specifying and verifying programs, it appears that the main emphasis has been on the use of the language for runtime assertion checking. This emphasis appears to be borne out by the choice, and implementation order, of tools developed for supporting work using ANNA. In particular, one area of research that is being looked into is the use of parallel processors to provide concurrent checking of specifications.

ANNA is being utilized in a number of research and development efforts for the specification of systems to be implemented in Ada. Since the ANNA semantics closely parallel those of Ada, its use in secure systems development would allow the system designers and implementors to use the same underlying language semantics for communication of the intended behavior of their specifications and programs. However, since the language appears to have been targeted to the runtime validation of program execution rather than pre-execution proofs of correctness, its applicability in secure systems development would be limited until a supporting infrastructure, both in terms of theoretical aspects of the language and in terms of automated tools, can be developed.

### TSL

TSL is a language for specification of the parallel aspects of Ada program execution. As with ANNA, the efforts appear to be focused on the runtime validation of TSL specifications. The model of computation of an Ada program used in analysis is that of a partially ordered linear sequence. The stream of such a sequence provides a thread of control, which may be analyzed to determine the proper execution of a program.

### HDL

HDL (Hardware Design Language) is an instance of VHDL (VHSIC HDL) that incorporates features of both VHDL and ANNA for hardware design. The major benefit to VHDL that will accrue through the synergy with ANNA is the ability to more accurately represent hierarchical system design, and to permit the use of axiomatic proof methods in order to perform design verifications.

## CURRENT TOOLS

This section describes the status of tools in the current development environment. The tools are referred to in several categories by the developers. The only category discussed here is that of the application tools, the ones with which the user interfaces. These tools have no direct bearing on the development of verified secure system implemented in Ada.

The Ada-ANNA Fabricator and Structure Editor is the planned user interface to all the tools in the integrated environment. As such, it operates on the DIANA Abstract Symbol Tree. These tools understand the syntax and structure of both Ada and ANNA, and are capable of inserting templates and filling in syntax information. The system has a graphics display with a mouse interface.

The ANNA Runtime Checker is the cornerstone of the current research efforts. It provides a capability for testing Ada programs for consistency with respect to ANNA specifications through runtime validation checks. ANNA annotations are parsed, stored in internal format, and pretty-printed as part of the Ada program that they are intended to validate. The checker is functional, with ports completed to several hardware architectures. A number of small examples have been run through the system, and some external organizations have used it as well.

The TSL Runtime System provides the same capability for parallel programs that the ANNA Runtime Checker provides for *sequential* programs. This is accomplished through a runtime monitor that accepts the execution stream of relevant events specified using TSL, and matches them against the sequence derived from the specification provided in the program. Errors are raised upon observance of inconsistencies.

### Directly Related Tools

This section describes some of the planned tools for supporting software and hardware development with the various languages described previously. Only those tools which are applicable towards verified secure systems are included.

The ANNA Specification Analyzer permits the interactive evaluation of package specifications during development. The specifications and program can be symbolically evaluated, and stopped at any particular point in the evaluation for investigation of existing properties of the system under evaluation.

This tool could be used during initial design and development of verified secure systems implemented in Ada to allow the designer to obtain a much greater understanding of the system specification, and the consequences of various decisions.

This tool would provide extensive support for the development of verified secure systems implemented in Ada. However, this project is considered a major and very longterm development effort, which is still in the planning stages, and does not appear to be a primary focus of the research effort. Such a tool appears to be at least three to seven years away from being available and useful.

The ANNA Static Checker would allow the checking of some annotations to be done prior to runtime through the use of static semantic analysis. This would allow some of the runtime checks to be discontinued, saving on execution time for the system under evaluation. This tool would provide an initial subset of necessary components for a verification system.

The ANNA Verifier will provide support for mathematical proofs of correctness between ANNA specifications and Ada programs. It is intended to support and be integrated into the software development process, support a number of development methodologies (although be particularly oriented to hierarchical development), and provide for both fully automatic verification and for stepwise theorem proving.

Indirectly Related Tools

This section describes some of the planned tools which are not applicable towards verification in Ada.

The ANNA Package Body Developer follows the Specification Analyzer and aids in constructing the package body. The specifications prepared in the Analyzer are used as specifications for the various bodies in the package implementation, and additional assertions, such as loop invariant assertions, are developed and added to the implementation.

The ANNA Runtime Testing System appears to be an extension of the current ANNA Runtime Checker. The major extension is the inclusion of reasoning tools to support the optimization of inserted runtime checks, a la McHugh.

This tool is intended to generate checks that can be run in parallel with execution of the underlying program to check tasking related software. In addition, it will permit the investigation of further applications of system parallelism by allowing tasks that run the checks to be placed on other processors.

The TSL Distributed Simulator-Checker is an extension of the current TSL runtime system to provide simulation of parallel (tasking) software specifications prior to implementation. Specifications of the interactions between tasks are specified in TSL, and the specifications are symbolically evaluated to determine consistency.

# APPENDIX C

# OVERVIEW OF SOFTECH'S ADA AXIOMATIC SEMANTIC EFFORT

SofTech has been working on an effort to define the problems and potential solutions to the development of an axiomatic semantic definition of the Ada language. The difference between an axiomatic semantic description of Ada and the definition of Ada given by MIL-STD-1815A is that the semantic description defines the behavior and interrelationships of the individual language constructs in such a way as to be used as the basis of a proof. The existence of a semantic definition of a language is necessary if a comprehensive verification technology is to be developed for that language. Any aspects of a language that are not rigidly, semantically defined are subject to varying interpretations by different compilers. Some of the Ada constructs which pose difficulties in verification have been left out of the semantic description. A list of the excluded constructs are address clauses, unchecked conversions, variables shared among tasks and subprogram calls that generate aliases.

To a large extent, most elements of a semantic description are handled at compilation time and need not be dealt with during verification time. It is important to realize that the actual verification environment is based on the semantic definition of Ada rather than the actual language and constructs which are not included in the semantic definition invalidate the verification process. The SofTech study concerns itself only with those constructs which are not dealt with at compilation time.

SofTech has developed the architecture for a verification environment based upon the formal semantic definition they have developed for Ada. This verification environment is based upon a modification of Ada -- the semantic definition. The description of the proposed verification environment is included in the following subsections.

## THE VERIFICATION ENVIRONMENT

To create an environment to verify Ada programs three languages were identified as needed to be defined. These languages are the source language, the assertion language, and the metalanguage.

The source language is based on the code to be verified. The source language builds upon the original Ada code by removing any and all ambiguities and overloaded constructs contained within the Ada code and specifying them in such a way as to make them unambiguous. An inconvenience of this language is that the Ada code which has been modified to become source language code tends to be verbose and difficult to read. A source language to Ada translator may be used to transfer code back to Ada to add enhanced readability. Aliasing is not allowed within the source language even though it is allowed within Ada.

Assertion languages, of which ANNA is an example, describe conditions which are expected to hold for formal verification. The most important quality of the assertion language is the availability for logical operators within the instruction set.

The proof rules are written in the metalanguage. The metalanguage is designed after Dijkstra's weakest liberal precondition function (wlp) wherein each function contains the parameters S and P where S is a statement and P is a formula. If S is true both before and after the wlp is executed then the formula P will be true after wlp is executed.

## IDENTIFIED PROBLEMS

SofTech identified a series of problems that arise while attempting to verify their axiomatic semantically defined Ada.

### Erroneous Execution

As described in the Ada Language Reference Manual (LRM) there are four types of program errors:

1) Those which are detected at compile-time and make the program illegal. These are of no concern to a verification environment. It is assumed that any program undergoing verification has passed through a compiler.

2) Those which are detected by run-time checks and raise exceptions. This error is discussed in Section 3.2.2.2.

3) Those which need not be detected at all, but whose occurrence makes execution of a program erroneous.

4) Those which need not be detected, but which cause the outcome of the program to be considered erroneous.

The third type of error is erroneous execution, although when this type of error causes execution problems, the programmer, rather than the program, is considered to be at fault. This type of error is considered to be outside the domain of verification.

Semantic definitions of programs fail to hold during an erroneous execution. The verification of the abstract package state corresponds to the modular verification of subprograms. As with the subprograms, abstract package state specifications and bodies are verified separately.

An erroneous program (an error of type four) is a result of poor specifications or incorrect programming, problems that are beyond the scope of present verification technology. State of the art verification technology is capable of verifying the consistency of a program only with regards to its specifications.

### Unpredictable Exceptions

Unpredictable exceptions are those exceptions that are raised from sources external to the program, e.g., DEVICE_ERROR. It is suggested that this type of exception may be handled by the creation of a logical predicate that is true as long as none of these "acts of God" occur. This predicate may be added to the preconditions for the occurrence of the event.

## Validation of Modular Sections of the Program

Ada is designed so that modular portions of Ada code are capable of being compiled separately. Verification of Ada programs should also be capable of being performed separately in a similar manner. To perform modular verification, semantic definitions must be made of the modular specifications and modular bodies. The verification of a module's body is not affected by the body of another module; only the semantic definition of the specification is considered. When a body is verified the preconditions established by the corresponding specification must be maintained.

A problem occurs during verification if the value of a parameter is dependent upon whether it is passed by reference or copying. If this possibility exists then the subprogram call is considered erroneous. This evaluation includes occurrences of this problem through aliasing.

Packages are verified modularly by the modular verification of an abstract package state which corresponds to the subprogram specification.

## Tasking Issues

Tasks, like packages, are treated as abstractions with an abstract state and a body being verified separately. Each task entry has a set of pre-postcondition pairs which must remain valid for verification to take place. Exceptions raised during tasking are considered "acts of God" and the raising of a tasking error invalidates the semantic description. In this way the issue of concurrency problems is avoided.

## Timing Issues

Timing issues, such as delay statements, wait statements, and the priority pragma are avoided by the SofTech semantic description.

## Optimization of Compilers

The optimization of code in Ada allowed by the Ada LRM presents a problem in formulating axiomatic semantics and subsequent verification. The modification of the Ada language by the compilers makes it impossible to properly define the required preconditions.

# APPENDIX D

# OVERVIEW OF MAVEN: THE MODULAR ADA VERIFICATION ENVIRONMENT

SofTech has proposed a set of requirements for tools to support the formal verification and validation of Ada programs on a modular basis. This set of requirements is known as the Modular Ada Validation Environment (MAVEN). This project is not being implemented and is meant to serve as a set of guidelines for an Ada Program Support Environment (APSE) that is to support formal verification and validation. SofTech makes the distinction between verification and validation by defining them as such: verification pertains to the formal proof of a piece of code through mathematical techniques, validation pertains to the determination of a level of confidence in the software in question through formal or informal methods.

## REQUIREMENTS

The functionality of MAVEN is based on the establishing of several requirements for the validation of Ada programs:

a) Formal proofs should be implementation independent. The behavior of a specific environment that contains a validated Ada compiler should not affect the performance of MAVEN.

b) The functional performance of MAVEN should be modular in nature. The modification of a modular section of a program should not require the re-validation of other modules of the same program as long as certain pre-postconditions of the modified module are not changed.

c) Separate modules of a program may be validated by different techniques. Different modules lend themselves to different validation techniques. While one module may be formally verified without complications, another module may require a less rigorous technique be used. This difference in techniques should not affect the overall validation of the software. Some of the possible techniques include formal and informal proof techniques, code walkthroughs, unit testing, and historical acceptance based on trusted performance.

d) While complete verification may be impossible with existing techniques, partial verification or validation, involving the proof of one or more properties of a module or program, is a useful indicator in providing a guide of correct behavior.

## TOOLS

To perform validation of a module of code it is suggested that a process similar to the compilation of a module of code be used. The syntactic specification of a piece of code should be validated before that piece of code's body. This allows other modules that call this module to be validated before the module is fully validated. The eventual validation of the body includes maintaining the constraints defined by the syntactic specifications.

Facilitating modular validation, an important construct with MAVEN, is the program library. Syntactic specifications are stored in the program library where they are accessible to module bodies being validated. Semantic specifications are stored in a validation library. The semantic specification for a module consists of a set of pre-postcondition pairs, one for normal termination and one for each exception that may be raised.

To facilitate consistency within the verification process, MAVEN imposes restrictions on the order in which units may be verified. Specifically a module's semantic specification must be entered into the validation library before any other module that references the module, or the module itself, may be considered verified. The order of verification of individual bodies or individual specifications is not considered, only that a specification must be verified before a separate body that refers to it may be verified. Similar restrictions apply to the re-validation of a program after modifications are made.

While the most powerful tools within MAVEN are the libraries (validation and program), there are more verification tools within MAVEN. Some of these tools are specification writer's assistant, a software retrieval tool and a suite of testing tools. The specification writer's assistant is a knowledge-based tool which, through interaction with the user, produces formal specifications. The software-retrieval tool implies validation upon reusable units of code assuming two conditions hold true:

a) The preconditions given in the design imply the corresponding preconditions of the reusable unit.

b) The postconditions given in the design imply the corresponding postconditions of the reusable unit.

## EFFECTS ON THE SOFTWARE LIFE CYCLE

While MAVEN is mostly applicable to the validation of software, it provides support for a wide variety of topics over the entire life cycle. Only those topics which are directly related to verification will be discussed here. Under MAVEN, verification is not limited to a single do or die process, support is given to verification of software throughout the life-cycle. The specification writer's assistant is useful in defining formally stated verifiable requirements. In addition MAVEN plays four roles in the design phase:

a) The validation library is used for storage of semantic specifications of each design module. Again the specification writer's assistant is used here.

b) During top level design module design plans are formulated and stored in the validation library.

c) MAVEN contains a catalog of reusable software including modules previously placed there by the user. These modules are most easily integrated if their formal specifications are stored in a library which is linked to the actual code through automated retrieval. As previously stated, re-validation of this code is not required if the syntactic specifications validate within the scope of the new program.

d) In the same way that standard Ada code is verified an Ada Program Design Language (PDL) is subject to validation through MAVEN. This enables the programmer to prove that top-level algorithms correctly meet the system specifications.

Logically, all verification techniques applicable to top level design are also applicable to intermediate and low level designs.

MAVEN may be used to assure that, when modifications are made to the program for maintenance, the initial specifications are not validated.

## APPENDIX E

## OVERVIEW OF SOFTWARE SAFETY

There are strong parallels between security and safety. In security, if software performs unpredictably national interests may be compromised. Software safety is typically utilized to protect human life in instances as varied as monitoring nuclear power plants to regulating implanted pacemakers to supporting air traffic control.

The question here is Ada-specific: Is the software safety approach applicable to Ada software that is intended to be trusted or secure? The two issues spawned by this question are the degree of similarity of security and safety, and the maturity and applicability of software safety vis a vis Ada. To address these issues it is necessary, first, to review software safety.

## DESCRIPTION OF SOFTWARE SAFETY

Software safety involves ensuring that software will execute within a system context without resulting in unacceptable risk. Many safety-critical applications will be written in Ada, and therefore it is useful to examine the Ada language with respect to safety issues. The DoD and other government agencies require that all safety-critical software be certified as having acceptable risk before it can be fielded. The ability to verify the safety of software written in Ada may make the difference between whether Ada features are used in safety-critical software.

To make these decisions, it is helpful to examine the types of requirements levied on safety-critical software. A general safety standard MIL-STD-882B: System Safety Program Requirements, was recently updated to include two tasks specifically related to software. One task requires Software Hazard Analysis "to identify hazardous conditions incident to safety critical operator information and command and control functions." This task involves performing and documenting "software hazard analysis on safety critical software-controlled functions to identify software errors-paths which could cause unwanted hazardous conditions" by examining "software and its system interfaces for events, faults, and occurrences such as timing which could cause or contribute to undesired events affecting safety." A second task, Safety Verification, requires that the developer "define and perform tests and demonstrations or use other verification methods on safety critical hardware, software, and procedures to verify compliance with safety requirements." On most non-trivial software, it will not be possible to verify safety to the degree required by using testing or demonstrations. This will mean that formal verification procedures of some sort will be required by those on whom this task is levied.

Two service-specific standards also require software safety analysis. An Air Force standard for missile and weapon systems, MIL-STD-1574A: System Safety Program for Space and Missile Systems, requires a Software Safety Analysis and an Integrated Software Safety Analysis (which includes the analysis of the interfaces of the software to the rest of the system, i.e., the assembled system). The Navy also has a draft standard, MIL-STD-SNS: Software Nuclear Safety, due to be released soon that requires Software Nuclear Safety Analysis (SNSA). This Navy standard will impose strict requirements on software design and verification that will severely affect the choice and use of the programming language for the software. If strictly enforced, these standards could also mean that some type of formal verification of safety will be necessary.

Achieving acceptable risk for systems controlled by software will require changes to the entire software development life cycle. The programming language used will most affect the design, implementation, and verification activities.

## Design and Implementation

Designing for safety requires early identification and separation of safety-critical functions. This allows design to focus on areas requiring particular and intense attention and to provide leverage for the verification activities by minimizing the amount of time and effort needed to verify and certify the software.

Software hazard analysis involves identifying software-related system hazards. It is performed early in the development process, usually prior to the specification of the software requirements, but after preliminary system hazard analyses have been performed. In general, software can cause problems through acts of omission (failing to do something required) or commission (doing something that should not be done or doing something at the wrong time or in the wrong sequence). Software hazards also may include failing to recognize a hazardous condition requiring corrective action or producing the wrong response to a hazardous condition.

Once the software safety requirements have been identified and specified, it is necessary to design the software to minimize risk. Software safety design analysis [Leveson (1986b)] is a procedure whose goal is to identify safety-critical items. The process can begin once a high-level design has been produced. Safety-critical items are software processes, data items, or states whose inadvertent occurrence, failure to occur when required, occurrence out of sequence, occurrence in combination with other functions, or erroneous value can be involved in the development of a potential hazard. Safety-critical items include erroneous program states and data items that could cause a hazard even if the function or algorithm is correct. Emphasis is placed on inputs from and interfaces with other components of the controlled system.

The results of the software safety design analysis are used in the detailed software design and implementation, especially with regard to minimizing the critical items, designing fault tolerance and exception-handling facilities, and ensuring that the critical items are isolated from the rest of the software and adequate "firewalls" built. It is also useful in planning load shedding and reconfiguration (e.g., determining exactly which modules and data items are absolutely necessary in a degraded (fail-soft) processing mode and determining the priorities that should be assigned to Ada tasks).

A safe software design includes standard software engineering techniques to enhance reliability, and special safety features such as interlocks, fail-safe procedures and design to protect against failure in other parts of the system including the computer hardware [Leveson (1986a)]. In general, the design features can be divided into two categories: (1) preventing hazards and (2) detecting and treating them.

Preventing hazards through design involves designing the software so that the occurrence of faults and failures cannot cause hazards. The basic idea is to reduce the amount of software that affects safety (and thus to reduce the verification and certification effort involved) and to change as many potentially critical faults into non-critical faults as possible. This may involve isolation of critical functions through modularization, the application of security techniques for authority limitation to ensure that critical items are protected from inadvertent activation or destruction, the use of programming language concurrence and synchronization features to ensure sequencing and to implement interlocks, etc.

Preventing hazards through design is difficult. In any certification arguments that are based on this approach, it will be necessary to prove that there is no way that the safety of the system can be compromised by faults in the non-critical software. One way to do this is to provide a programming language whose semantics ensure that the hazards are prevented. Ada has many features that will help here including strong typing, tasking, abstract date types, and exceptions.

Prevention of hazards is difficult and tends to involve reduction of functionality or design freedom. The alternative is to attempt to detect and treat hazards. Ada exception-handling provides a mechanism to assist in software fault-detection. However, it has been found to be difficult to formulate the appropriate exception conditions [Leveson, Knight, Cha, and Shimeall(1963)]. In terms of safety, it is possible to use the information obtained through the software hazard analysis and the software safety design analysis to guide the content and placement of the exception conditions.

## Verification and Certification of Safety

If the design is carefully done, verification and certification of the safety of software should be greatly simplified. The most costly procedures need be performed only on the modules that have been determined to be so critical that testing and other assurance procedures alone will not suffice to ensure acceptable risk. The verification procedures also need to ensure that the detailed design features related to safety-critical items and exception-handling have been correctly implemented and that the assumptions and models upon which the analyses have been based are correct. Because Ada language features can actually be used to ensure many of these required design features, the certification requirements will reduce to arguments about he correctness with which these features have been implemented in the Ada compiler used.

This raises some interesting questions with regard to certification of Ada compilers for safety-critical applications. Certification of the compilers for such applications requires a much higher level of assurance than provided by current Ada compiler certification procedures. Even compilers for languages such as Jovial require years of verification and certification to convince those contracting for safety-critical software that there are absolutely no faults contained in the compilers and that they will, with absolute certainty, produce correct and safe code. Few such certified compilers exist at all, which is one of the reasons safety-critical software is often written in assembly language (it is much easier to certify an assembler). It is doubtful that any complete Ada compiler could pass these stringent procedures. However, it may be possible to build compilers that do not attempt to handle all Ada language features. It will also probably be necessary to build compilers that do not require that the entire Ada run-time support facilities be resident in memory for all programs to run. It will be next to impossible to prove that a large runtime support system is correct and that it will not affect the correctness and safety of the application software. It will be necessary to write compilers that allow run-time facilities to be excluded if the particular features that it handles are not used in the application programs. Without this, it is doubtful that Ada can be used in safety-critical applications for embedded systems.

## APPLICABILITY TO SECURE ADA SOFTWARE

It is important to understand that verification of safety is different from the usual verification of correctness [Leveson (1983), Leveson (1986a)]. The basic goal of safety verification is different than that of correctness. We will assume, by definition, that the correct states are safe (i.e., that the designers did not intend for the system to have accidents). The incorrect states can then be divided into two sets -- those that are considered safe and those that are considered unsafe. Safety verification attempts to verify that the program will never allow an unsafe state to be reached (although it says nothing about incorrect but safe states). Since the goal is to prove that something will not happen, it is useful to use proof by contradiction. That is, it is assumed that the software has produced an unsafe control action, and it is shown that this could not happen since it leads to a logical contradiction.

Although a proof of correctness should theoretically be able to show that the software is safe, it is often impractical to accomplish this because of the sheer magnitude of the proof effort involved and because of the difficulty of completely specifying correct behavior. In the few safety proofs on real software that have been performed [Leveson and Harvey (1983), McIntee (1983)], the proof appears to involve much less work than a proof of correctness (especially since the proof procedure can stop following a software path as soon as a contradiction is reached). Also, it is often easier to specify safety than complete correctness, especially since the requirements may be mandated by law or government authority, as with nuclear weapon safety requirements in the U.S. Like correctness proofs, the analysis may be partially automated [Rolandelli, Chimeall, Leveson (1986)], but highly skilled human help is required.

The degree of applicability of software safety approaches to secure software is a function of the ability to clearly identify the security requirements and to then isolate the code affected by these requirements. If a well defined set of security requirements exists and these can be partitioned into a limited number of modules then the software safety approach is applicable. It must be noted that with this approach, a large percentage of the code is developed using traditional approaches. If the intent is to know exactly what the software will and will not do this approach is not applicable; if, however, the intent is to assure that a set of requirements that are isolated into a limited number of modules are satisfied, then the approach is promising.

Relative to Ada, Software Fault Tree Analysis (SFTA), which has many of the features required for verification of safety, has been defined at least partially for the language [Leveson and Stolzy(1983)]. The technique can handle all Ada features that also occur in Pascal along with the Ada rendezvous. Work is presently being done at the University of California, Irvine to extend the technique to as much of Ada as possible (and to define those Ada features that might need to be avoided if SFTA is to be used). An automated tool that would take annotated Ada code and automatically produce the fault tree verification is currently under design and development, but will not be ready to be used for several years. Since the results can be checked by hand, it should be possible to certify such a tool for use on safety-critical code.

## REFERENCES

Jahanian, F. and Mok, A.K. "Safety analysis of timing properties in real-time systems," IEEE Trans. on Software Engineering, vol. SE-12, no. 9, Sept. 1986, pp. 890-904.

Leveson, N.G. "Verification of safety," Proc. Safecomp '83, Cambridge, England, Sept. 1983.

Leveson, N.G. "Software safety: Why, what, and how," ACM Computing Surveys, vol. 18, no. 2, June 1986a.

Leveson, N.G. "Building safe software," Proc. Safecomp '86, Sarlaet, France, Oct. 1986b.

Leveson, N.G. and Har[illegible]y, P.R. "Analyzing software safety," IEEE Trans. on Software Eng[illegible]neering, vol. SE-9, no. 5, Sept. 1983, pp. 569-579.

Leveson, N.G., Knight, J.C., Cha, S.D., Shimeall, T. "An Empirical Study of Software Error Detection using Self-Checks," submitted for publication.

Leveson, N.G. and Stolzy, J.L. "Safety analysis of Ada programs using fault trees, IEEE Trans. on Reliability, vol. R-32, no. 5, Dec. 1983, pp. 479-484.

Leveson, N.G. and Stolzy, J.L. "Safety analysis using Petri nets," IEEE Trans. on Software Engineering, in press.

MIL-STD-882B. System Safety Program Requirements, 30 March 1984, U.S. Department of Defense.

MIL-STD-1574A (USAF) System Safety Program for Space and Missile Systems, Dept. of Air Force, 15 August 1979.

MIL-STD-SNS (NAVY). Software Nuclear Safety (Draft) June 1984.

McIntee, J.W. Fault Tree Technique as Applied to Software (SOFTREE), BMO/AWS, Norton Air Force Base, CA 92409, 1983.

Rolandelli, C., Shimeall, T., and Leveson, N.G. "Software Fault Tree Analysis Tool: User's Manual," Technical Report, Information and Computer Science Dept., University of California, Irvine, 1986.

Vesely, W.E., Goldberg, F.F., roberts, N.H., and Haasl, D.F. Fault Tree Handbook, NUREG-0492, U.S. Nuclear Regulatory Commission, Jan. 1981.

## APPENDIX F

## OVERVIEW OF THE IBM CLEANROOM

The intent of IBM's "cleanroom" approach to software development is to prevent software defects during development rather than to detect and remove defects after development. The approach is to use a combination of human verification and statistical testing and to assess success of the process by utilization of reliability estimates. The human verification process is based on reasoning about sets, functions and relations of abstract states through use of denotational semantics. This approach enables abstraction to higher levels and, therefore, permits reasoning about both 100 line modules and 100,000 line systems. An underlying assumption is that specification writing, verification writing, and code writing are full partners and that each of these activities may affect tne other two. One conspicuous outgrowth of the approach is the attention that both individual programmers and programming teams give to the design and verification of software.

Whether this approach is applicable to developing secure software in Ada does not center on the Ada language. The questions are does this relatively informal approach satisfy the stated and intended software security requirements? Is human verification, which is both fallible and flexible, acceptable as an alternative to automated verification? Is the process of "cleanroom" both mature enough and adequately articulated to be transitioned to other organizations? The effect of Ada on the answers to any of these questions is secondary at most.

## DESCRIPTION OF CLEANROOM

A primary principle of "cleanroom" that supports the defect prevention philosophy is counterintuitive, at least initially: testing and debugging are prohibited until software is released for independent, user-generated statistical testing. This principle is an affirmation that the software will be developed correctly, and therefore unit testing and debugging are unnecessary. The debugging process is replaced by human verification combined with statistical testing.

Human verification is performed at each level of the stepwise refinement of a structured specification. A structured specification is a formal statement of a specification as a relation. This relation is given as ordered pairs of potential inputs and acceptable outputs. The initial specification is decomposed into a nested set of subsets that are used to define the initial and incremental releases. Due to the nesting, each incremental release includes all the specifications of previous releases.

The testing performed prior to release is customer-defined statistical testing. The customer generates a use profile for the subset of specifications covered by each release. The testing is designated to establish reliability levels for the software. Since test suites are developed based on anticipated usage, translating the performance during testing into anticipated reliability during use is fairly straightforward.

## Human Verification

The verification used in "cleanroom" is functional verification and the only automated tools used are word processors. Functional verification utilizes the denotational semantics and rules of sets, functions and relations to reason about specifications as they are decomposed and refined.

Initially, a structured specification exists at a very high level. This specification is given as a relation, a set of ordered pairs of anticipated inputs paired with acceptable outputs. This specification is refined by replacing pairs of the relation with more concrete rules for determining the output element of the pair. Two processes are applied during refinement. One is the decomposition of the specifications into nested subsets. The subsets of specifications are used to define incremental deliveries where each delivery is an expansion of the previous delivery; the specifications covered by the new delivery are a superset of those covered by the previous delivery. In addition to being broken into nested subsets, the specifications are refined in detail. This evolution of specifications leads to specifying each sequence of statements, each iteration (loop) statement, and each selection (IF. . .THEN) statement. The specifications are developed prior to the code. Obviously, large volumes of specifications are generated.

The verification process takes place in parallel with the specification development. The reasoning is done using denotational semantics based on the set theory. From the initial structured specification through detailed specification that will be used to generate a small sequence of code, specifications are given as relations. (If the output for any given input is unique, the relation is a function.) As the specifications are decomposed and refined, more specific rules for calculating the relations or functions are generated. Mathematical variables are introduced and, eventually, program variables are defined and introduced into the specifications. It is important to note that when the reasoning goes from the more detailed specifications to the higher level specifications, that the abstraction process replaces the program variables and mathematical variables so that the verification documentation remains manageable from the perspective of detail.

The establishment of program correctness with respect to a specification is finalized after the program has been written. The specification for the program has been given as a relation. The program computes a function, since any input will calculate a unique output. Each small piece of code is specified by a function, typically given as a rule in terms of program variables. To calculate the meaning of larger sections of code, the functions of the subsections of the code are composed. As the sections of code functionally described get larger through composition, the specific function rules are replaced by higher-level descriptions which no longer utilize the program variables. Once a function for the entire program has been established it is compared to the relation of the specification.

The program is considered correct with respect to the specification if for each input in the specification relation, the program calculates an acceptable output.

This can be represented quite concisely mathematically. Let the specification relation be denoted by S. S is a set of ordered pairs. Let the function calculated by the program be denoted by P. P is also a set of ordered pairs. The domain (S) is the set of all first elements in the ordered pairs of S, and the domain ($S \cap P$) is the set of first elements of ordered pairs that are in the intersection of S and P. P is correct with respect to S if and only if

$$\text{DOMAIN } (S \cap P) = \text{DOMAIN}(S).$$

This assures that each input element in S is paired with an acceptable output element by the function P.

The verification text generated is not a full proof. The text is the design of a proof that is meant to be inspected by other designers and judged for its adequacy.

Statistical Testing

This process of verification is fallible for two reasons. First, it is a human activity and humans are fallible. Second, proof outlines may not expose subtle and critical errors in reasoning. For these reasons, the "cleanroom" complements the human verification with statistical testing.

Statistical testing is based on user-generated profiles of the use of the software being produced. Test suites are developed which reflect this profile. This method of testing encourages covering a large portion of the specification input space and also utilizes the design hypothesis, which is still intact due to the nature of the development process.

Once testing is complete, reliability estimates can be made. These estimates are based on the performance of the software during statistical testing.

APPLICABILITY TO SECURE ADA SOFTWARE

"Cleanroom" is an approach to enhance and estimate software reliability. The human verification process embedded in "cleanroom" is an attempt to ensure that the output of the program is acceptable as defined by the specification. If the specification accurately states what the program is to do, the human verification process attempts to assure the program performs as it is desired to perform.

The benefits of "cleanroom" include the attention given to the design and verification process by individual programmers and by programming teams. Also, the design and the design hypothesis are preserved through this process. The verification and inspections happen early in the life cycle so that evolving designs that are hard to verify may be altered.

Harlan Mills, architect of the "cleanroom" approach, has made Ada-specific recommendations for verification [Mills 1986A]:

- o Transfer verification technology to people as well as machines -- people are more fallible but more flexible.
- o For human verification in an Ada environment to be successful, a verification language and processing facilities are necessary.
- o Include in the verification or specification language a proposition type whose members are proofs or proof outlines, for the proposition.

The appealing features of applying "cleanroom" to certification of secure software include that the process goes from early in the specification phase through coding and testing and therefore, beyond design verification; it is a formal method, although somewhat informally applied; and it is somewhat language-independent. It does however, require training talented individuals, and the process is only as good as each specification application.

## REFERENCES

Mills, Harlan, "Human Verification in Ada." Presented at the Third IDA Workshop on Ada Specification and Verification, Research Triangle Park, NC., May 14-16, 1986.

## APPENDIX G

## USE OF AUTOMATIC PROGRAMMING AND IV&V TO INCREASE CONFIDENCE IN SOFTWARE

## AUTOMATIC PROGRAMMING

Automatic programming has been the hope of many computer scientists since the beginning of the field. The term has been applied to a widely varying set of technologies, but they always seem to be what is just over the horizon.

In the fifties, FORTRAN was called an automatic programming tool. In many ways it did automate the programming of arithmetic formulae. Many errors were eliminated, optimizations could be done by the translating software, and consequently higher confidence could be placed in the result. There was still considerable opportunity for error -- misused operator precedence and parentheses for example. These kinds of errors could not be caught by the compiler because they were correct programs, just not the ones intended. The attempt to answer this issue drew things in two different directions. One was toward testing and the other was toward an even higher level of specification of the program.

Testing was basically to show the actual performance of the software. These results could be compared with the expected results and thereby determine if the program was correct. The development of these tests required as much, if not more, analysis and development work as the program itself. It was also clear that testing could uncover only a limited number of errors. In particular if the original problem was misunderstood, it was possible for both the programming and testing teams to do their jobs wrong. Sometimes they would catch each other, but sometimes they would make consistent misinterpretations.

In many situations the original problem statement and its analysis would be done by a single team and then passed to developers and testers. If the original analysis was wrong then no amount of work by the testing team would reveal it.

This motivated the other approach which was to move to higher level languages which would be closer to the original problem and hopefully understandable by the people with the problem to be solved. By using a higher level language, the number of interpretation (and thereby potentially error causing) steps could be reduced. This is also some of the motivation behind prototyping. The general hope is that written at the proper level, it will be possible for the original requestors to determine if it is the correct program. Then the actual program could be automatically derived through a series of transformations, program generation, and compiling steps.

Much of the work on program proving is closely connected to this approach. By putting assertions about the program into the code itself, it is hoped that the program can be automatically proved to conform to these assertions, and it is also hoped that these assertions will be more directly linkable to the requirements and specifications of the original problem.

Because of the close linkage of these two approaches, it is unlikely that one will succeed without the other. Current approaches in automatic programming are along the lines of automating the generation of programs starting from very high level specifications. These very high level specifications are very much like the high level assertions used in the program proving and verification approach.

Research and development work going on now should emphasize some of the common features of the two approaches and start providing the programmer tools and interfaces that would be useful -- for example, some kind of advanced editor that would help build high level assertions and link them to the high level specifications and requirements that are supposed to be answered by the code under development. This kind of tool could be immediately useful in the context of traditional IV&V (Independent Verification and Validation) where requirements, specification and code must be linked and also useful in developing code which had a high probability of fitting into any automatic verification environment developed in the future.

Some of the work on program transformations has already been applied in unstructured to structured code converters and in recursion removal optimizes. Many other transformations could be used to assist programmers. The programmer's assistant kind of work is really just a high level program building tool, but it still requires an expert programmer and expert in the tool to use it effectively.

## INDEPENDENT VERIFICATION AND VALIDATION (IV&V)

In IV&V (independent verification and validation), the word "verification" basically means checking between stages in the software development process to be sure that all items have been correctly handled. For example, that each of the points in the specification is accomplished by a portion of the design, that no functions exist in the design that are not explicitly required in the specifications, and that all the connections between the requirements specification and the design have been properly documented.

This approach is very useful in large systems where it is very easy for things to get lost. There have also been some automated systems which provide various kinds of cross reference lists linking specification sections to design elements. These can also be used to link testing objectives to specific modules, chec' for the modularity and separability of the design, and point out related areas during maintenance activities.

On government contracts the IV&V process can also be used to have separate contractors checking on each other about the details in the specifications and designs (usually a pretty good idea). There is however a problem in that the organization managing the acquisition of the computer software is usually separated from the initial requestor and from the eventual user. That means there will be continual interpretations of what is wanted that may, or may not, match exactly the actual goals.

This approach has also begun to use various metrics and models to predict the complexity and error proneness of large software developments. There are also formal procedures for software audits and so forth. These topics are well covered in many current software engineering text and guide books.

This approach is neglected by more formally oriented computer scientists because of its being generally a management technique. The cleanroom and software safety approaches have their roots in these now classical approaches to software management.

When it comes down to human decisions as to whether some software performs correctly, there will always be a need for management control and tracking of the development and review process. These classical techniques will be important no matter how much we are able to automate high level specifications, automatic derivation of programs and formal verification.

These techniques should be adapted to use the emerging formal specification tools, automatic cross references between stages in program development, and the limited formal verification technologies that are now becoming available. In particular code needs to be developed with eventual verification in mind. These management techniques can be used to assure that the software and its internal conditions and assertions are consistent.